

前言

《从入门到精通》系列图书是专门为编程初学者量身定做的一套编程学习用书，由龙马创新教育研究室策划，魔乐科技（MLDN）软件实训中心编著。

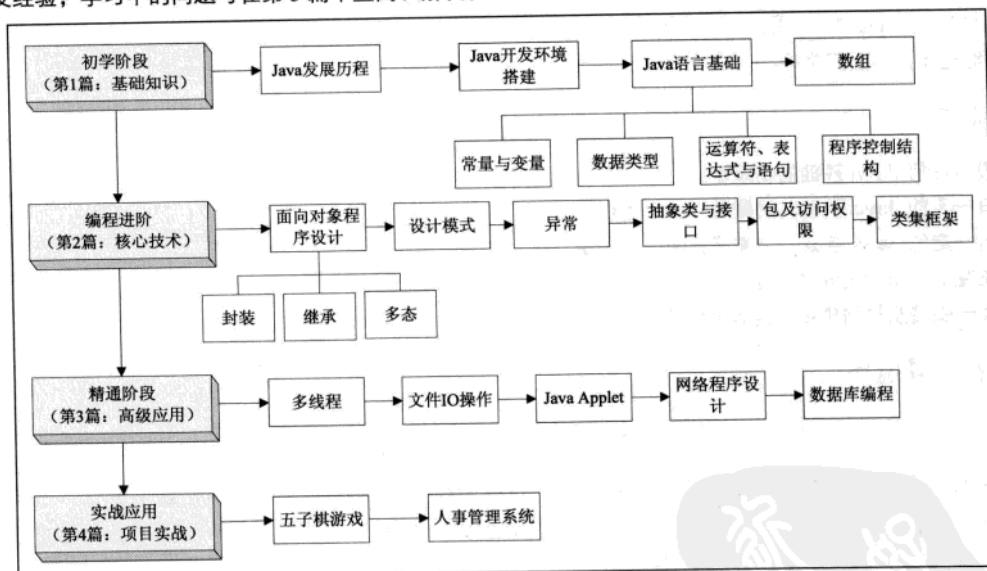
本书专门为 Java 初学者和爱好者打造，旨在使读者学会、掌握和能够进行项目开发。当您认真系统学习本书之后，就可以骄傲地说——“我是一位真正的 Java 程序员了！”，即使目前您还是初学者。

为什么要写这样一本书

古人云：不闻不若闻之，闻之不若见之，见之不若知之，知之不若行之。实践对于学习知识的重要由此可见一斑。理论知识与实践经验的脱节，恰恰是很多 Java 图书的写照。从项目开发经验入手，结合理论知识的讲解，便成了本书的立足点，也转化成了对本书作者的要求。我们的目标就是让初学者、应届毕业生快速成为一个初级程序员，拥有项目开发经验，在未来的职场中有一个高的起点。

Java 学习最佳途径

本书以学习 Java 的最佳结构来分配章节，前 3 篇可使您掌握 Java 的编程知识，第 4 篇可使您拥有项目开发经验，学习中的问题可在第 5 篇中查阅、解决。



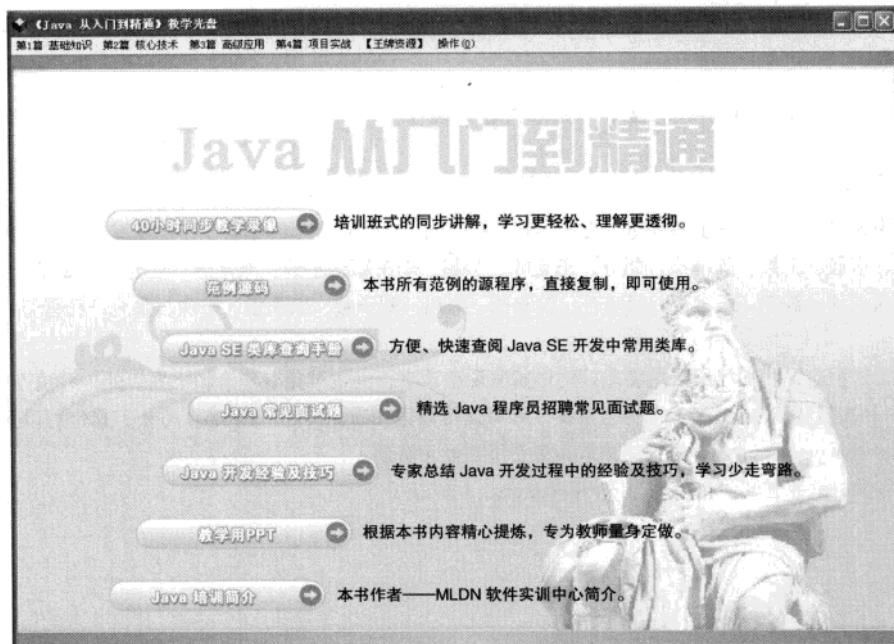
本书特色

■ 零基础、入门级的讲解

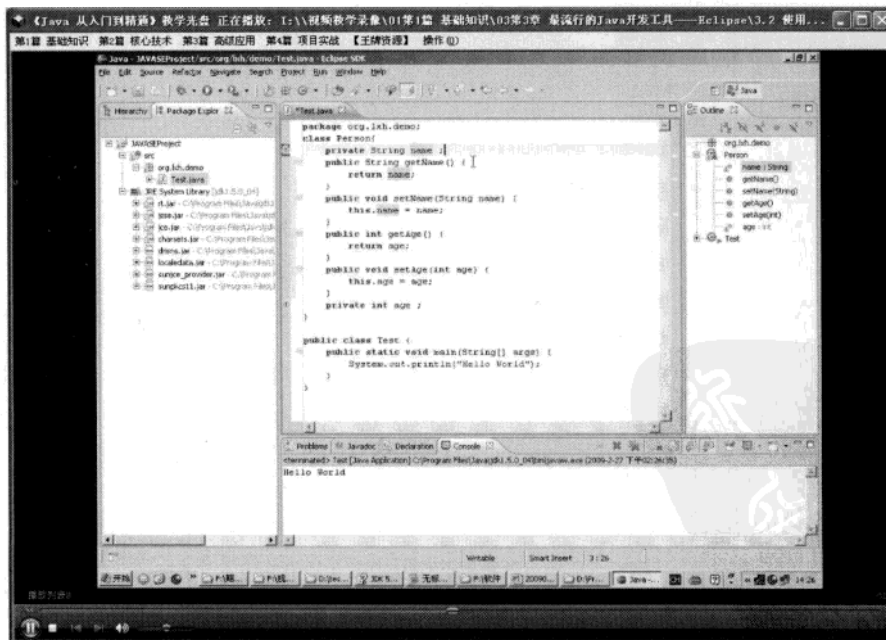
无论您是否从事计算机相关行业，无论您是否接触过 Java，无论您是否使用 Java 开发过项目，您都能从本书中找到最佳起点。

■ 超多、实用、专业的范例和项目

本书结合实际工作中的范例逐一讲解 Java 的各种知识和技术，项目开发篇中更以 2 个项目的实战来总结本书所学，使您在实战中掌握知识，轻松拥有项目经验。



- ④ 单击【40 小时同步教学录像】按钮，在右侧弹出的菜单中依次选择相应的篇、章、录像名称，即可播放本节录像。



- ⑤ 单击主页面中的其他按钮，则打开相对应的文件或文件夹。以上这些内容也可以通过选择菜单栏中的相应菜单命令来实现。

- ⑥ 单击菜单栏中的【王牌资源】，在弹出的菜单中选择王牌资源的名称，即可打开相应的王牌资源电子书。
- ⑦ 光盘使用详细说明请参阅光盘“其他内容”文件夹下的“光盘使用说明”文档来查看。

创作团队

本书由龙马创新教育研究室策划，魔乐科技（MLDN）软件实训中心编著，参加编写和资料搜集的人员有孔万里、李震、王果、陈小杰、胡芬、王金林、彭超、李东颖、左琨、邓艳丽、任芳、王杰鹏、崔姝怡、左花苹、刘锦源、普宁、王常吉、师鸣若、钟宏伟、陈川、刘子威、徐永俊、朱涛、张允、杨雪青、孙娟和王菲等。

在编写过程中，我们尽所能地将最好的讲解呈现给读者，但也难免有疏漏和不妥之处，敬请不吝指正。若您在学习中遇到困难或疑问，或有何建议，可写信至信箱 march98@163.com。另外，您也可以登录我们的网站 <http://www.mldnjava.cn> 进行交流以及免费下载学习资源。

责任编辑的联系信箱：liuhao@ptpress.com.cn。

Java 学习攻略

一、学习目标

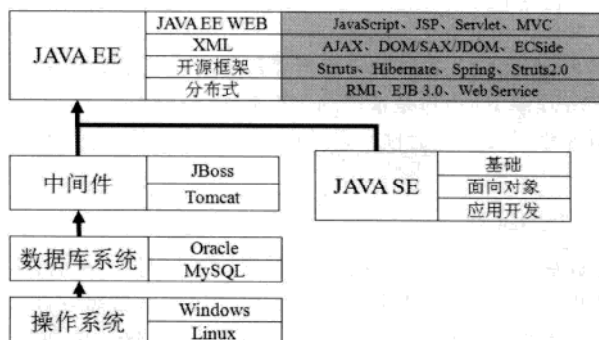
学习 Java 不仅是学习一门语言，更多的是学习一种思想、一种开发模式。而且从事软件行业的工作人员，掌握了 Java 语言，可以让自己日后的事业发展更加顺利。

在众多企业开发平台横行的今天，Java 语言以其简洁的语法、众多的厂商支持，成为了众企业平台开发的首选语言，掌握了 Java 也就相当于掌握了众多厂商提供的产品，从而为自己的事业发展创造更多机会。

不仅如此，Java 语言由于体系完整，所以 Java 开发人员可以轻松转入到手机开发、.NET、PHP 等语言的开发上，以后也可以更快地跨入到项目经理的行列之中。

二、企业平台

下图列出了企业开发平台的组成部分及其与 Java 技术的对应关系。



在企业进行开发的读者可以发现，所有的开发都是通过中间件来完成程序和数据库间操作的，所以在整个企业开发环境中，数据库是一个重要的操作，而如果想让操作更加合理，就要有很强的程序逻辑和良好的系统设计能力。

三、Java 与 Java EE 间的关系

Java 是一种企业平台的开发技术，里面有众多的技术，使用 Java 开发的企业技术称为 Java EE，所以在这之前每个读者必须首先搞清楚 Java SE 和 Java EE 的关系。



从上图中可以清楚地发现，Java SE 是 Java EE 的必要组成部分，这也就是为什么在学习 Java

EE 之前要读者一定要有扎实的 Java SE 基础了。

四、Java SE 的核心技术

Java SE 核心部分主要有四个，是日后直接与 Java EE 开发有关的。

1. 面向对象

主要是要求读者建立好面向对象的概念，并可以灵活运用这些概念进行程序的分析。可以说 Java SE 核心就是面向对象的开发，里面涉及的概念较多，但是所有的概念最终都是为接口和抽象类服务的，而所有 Java 开发中涉及到的各个设计模式，实际上也是针对于接口和抽象类的应用的。

2. 类集框架

大部分的读者都应该听过或学习过数据结构，虽然从大学就开始学习《数据结构与算法》，而且大部分的数据结构都是以 C 语言为基础讲解的。类集就相当于 Java 实现的数据结构，包括树、链表、队列等，如果只是进行应用层次开发的读者没有必要深入了解其内部的操作，但是必须清楚掌握其核心的接口和各个操作类。

3. IO 操作

对于大部分的自学者而言，IO 操作也许是最难过的一个坎了，如果没有很好的理解抽象类和接口的概念，要想充分理解 IO 操作是很难的。在整个 IO 操作之中，完整体现了 Java 语言多态性的设计思想，而且在学习 IO 的时候必须始终把握住一个原则：“根据使用的子类不同，输入输出的位置也不同”。

4. Java 数据库操作 (JDBC)

JDBC 首先并不能算是一门技术，更多的应该算是一种服务——Java 提供的数据库访问服务，里面提供了大量的操作接口，各个数据库生产商根据这些接口实现自己的数据库操作。从面向对象的设计上来看，用户完全没有必要考虑其接口是如何实现的，只需关心如何使用这些接口即可。从现在的项目开发中，大部分的开发都是基于数据库的。

虽然核心是以上四个，但是对于程序开发，读者应该建立起一些基本的程序思路。对于初学者来讲，应该将本书第一部分的知识打牢，要明白基本的循环操作。例如，打印正三角形、九九乘法表、冒泡算法等都是一些基本的要求。在学习程序的开发上没有任何的捷径，需要进行反复的代码训练才能将代码运用明白，就好像练习武功一样，只有坚持练习，才能让自己的动作收发自如。我在讲课的时候跟学生讲过，我的脑子里不会记住任何的代码，包括讲课中我也没有记住任何的代码，只是我敲熟了，用的时候代码就自然而然地写出来了，所以只有勤学苦练才能牢固地掌握编程语言。世界上或许有程序天才，但是我自己承认我和大部分人一样，都是普通人，都是一步一个脚印走出来的。

五、企业平台开发架构

在 Java 企业平台开发中，有两套开发架构——标准开发架构和开源架构。

标准开发架构就是使用了 SUN 提供的标准 Java EE 开发技术，使用 JSP/Servlet、EJB 进行开发的，如图所示。

我在

1

- 去重:

(5) 类与对象、类的定义结构、构造方法、private、static、this、引用传递、内部类，在本章一定要掌握简单类的开发方法。

(6) 类的扩展、super、final、抽象类、接口、多态、实际分析。

(7) 异常的产生原因、标准异常的处理语句格式、throw、throws、assert

(8) 包的作用、package、import、系统常见包、四种权限的关系

(9) Java 常用类库。在学习的时候不要花太多的经历去记住类的使用方法，关键是要学会如何查询文档，Java 提供的系统类太多了，每个人是不可能全部记下来的，用到何种功能大概可以想起来，之后通过文档查询其具体用法就行了，没有一个人可以将所有的类库都背下来，只需要将常用的灵活掌握即可。例如：StringBuffer、垃圾收集、Date、SimpleDateFormat、正则、比较器等

(10) Java IO, 面向对象的核心体现, File、OutputStream、InputStream、Writer、Reader、Serializable 等

(11) Java 类集的作用及使用，Collection、List、Set、Map、Iterator 等核心接口的使用

(12) MySQL、SQL 命令、Statement、PreparedStatement、ResultSet、事务处理，因为本书不是一本专门讲解数据库的书籍，所以在讲解的时候只是介绍性的讲解了部分的 SQL 语句。

2. 理解掌握内容：掌握其运行的基本效果，具体的代码可以不用全部灵活掌握

(1) 多线程：运行形式、两种实现方式及区别、同步及死锁

(2) 泛型：泛型更多的是应用在类集的概念上，所以本章只需要掌握泛型的使用形式即可

(4) 枚举：enum 关键字，如果没有习惯于使用枚举开发的人员，则本章只需要了解即可。

(4) Java 反射机制：重点部分就是在于如何取得 Class 对象，并且进行对象的实例化操作。

(5) Annotation：在 EJB 3.0 中使用较多，纯粹的 Java 开发主要就是三个内建的 Annotation。

(6) Eclipse 开发工具：开发工具本身不能算是重点，在本书讲解中基本上都很少使用到开发工具，读者一定要记住：只要程序会编写了，则开发工具也就自然会用了。

最后希望每一位读者都可以学有所成，因为从我个人的学习来看，只要学会了 Java，则.NET 或 PHP 都可以很容易学会，如果你自己本身还喜欢游戏的话，也可以学学手机的 Java ME 开发，实际上也都是很容易的。从一点一滴积累，现在就开始努力吧！

目 录

如何学习Java

第1篇 基础知识

开启 Java 之门。

第1章 初识庐山真面目——Java 6 2



视频教学录像：1小时21分钟

千里之行，始于足下。掌握一门编程语言的最好方法就是——亲自体验，本章将从零开始带领你一步步走进Java编程世界，指导你编写出第1个Java程序。

1.1	Java的历史	3
1.2	Java的现状	4
1.2.1	Java技术分支	4
1.2.2	Java语言的跨平台性	5
1.3	Java的特点	5
1.3.1	Java语言的优点	5
1.3.2	Java语言的关键特性	6
要进行Java开发，首先就要安装好开发工具，本节将讲解最新开发工具——JDK 1.6.0_17多国语言版的安装。		
1.4	安装Java开发工具箱——JDK 1.6.0_17多国语言版	7
1.5	磨刀不误砍柴工——配置开发环境	8
1.6	享受安装成果——编写第1个Java程序	9
1.7	classpath的指定	10
1.8	探秘Java虚拟机(JVM)	11
1.9	练一练	11
1.10	跟我上机	12

第2章 再识庐山真面目——简单的Java程序 13




视频教学录像：8分钟

Java的基本框架部分可以由一个简单而完整的例子来讲解，通过这个例子你将会对Java的开发有更切身的体会。

2.1	一个简单的例子	14
2.2	感性认识Java程序	15
2.2.1	认识Java程序的框架	16

2.2.2	认识标识符	17
2.2.3	认识关键字	17
2.2.4	认识注释	18
2.2.5	认识变量	18
2.2.6	认识数据类型	19
2.2.7	认识运算符和表达式	19
2.2.8	认识类	20
2.3	程序的检测	20
2.3.1	语法错误	20
2.3.2	语义错误	21
2.4	提高程序的可读性	22
	要想成为一名专业的开发人员，一开始就要养成良好的开发风格，遵循代码书写规则可以使你事半功倍。	
2.5	练一练	23
2.6	跟我上机	24


第3章 最流行的Java开发工具——Eclipse 25

 视频教学录像：38分钟

图形界面的开发工具使开发过程变得更有趣和直观，而附加的功能强大的插件使我们有更多理由选择Eclipse作为Java开发工具。

3.1	认识Eclipse开发工具	26
3.1.1	Eclipse概述	26
3.1.2	Eclipse的安装、设置与启动	26
3.1.3	Eclipse 工作台	28
3.1.4	Eclipse 菜单栏	28
3.2	使用Eclipse开始工作	30
3.2.1	创建Java项目	31
3.2.2	创建Java类文件	31
3.2.3	在代码编辑器中编写Java程序代码	33
3.2.4	运行Java程序	34
3.3	在Eclipse中调试程序	35
	在Eclipse中调试程序将变得再简单不过，一旦程序出错，你可以直接定位到出错行，快速清除程序bug。	
3.4	练一练	37
3.5	跟我上机	37

第4章 最常用的编程元素——常量与变量 38

 视频教学录像：14分钟

在程序运行过程中,有两种数据——固定的和变化的,就是常量与变量。掌握本章讲到的最常用的编程元素将有助于接下来的学习。

4.1	常量	39
4.1.1	声明常量	39
4.1.2	常量应用示例	39
4.2	变量	39
	在Java中对变量的命名有相应的规则,按照本节讲到的命名规则可以使你更容易编写出成功的程序,少走许多弯路。	
4.2.1	声明变量	40
4.2.2	变量的命名规则	41
4.2.3	变量的作用范围	41
4.3	练一练	42
4.4	跟我上机	43

第5章 不可不知的数据分类法——数据类型 44



视频教学录像: 21分钟

熟练使用数据类型是学好Java语言的基础,掌握数据类型后才能以此为工具实现更高级的功能。

5.1	整数类型	45
5.1.1	byte类型	45
5.1.2	short类型	46
5.1.3	int类型	47
5.1.4	long类型	47
5.2	浮点类型	48
5.2.1	float类型	48
5.2.2	double类型	49
5.3	字符类型	50
5.4	布尔类型	51
5.5	数据类型的转换	52
5.5.1	自动类型转换	52
5.5.2	强制类型转换	53
5.6	专题研究——基本数据类型的默认值	54
	本书专门归纳出所有基本数据的默认值,供开发人员参考使用。	
5.7	练一练	55
5.8	跟我上机	55

第6章 最重要的编程部件——运算符、表达式与语句 56




视频教学录像: 1小时18分钟

由运算符、表达式到语句，构成了Java语言的最基本部分，无论多么大型的软件，都是由这些重要的编程部件组成。

6.1	运算符	57
6.1.1	赋值运算符	57
6.1.2	一元运算符	58
6.1.3	算术运算符	59
6.1.4	关系运算符与if语句	61
6.1.5	递增与递减运算符	62
6.1.6	逻辑运算符	63
6.1.7	括号运算符	65
6.1.8	运算符的优先级	65
6.2	表达式	66
6.2.1	算术表达式	68
6.2.2	关系表达式	69
6.2.3	逻辑表达式	69
6.2.4	条件表达式	70
6.2.5	赋值表达式	71
6.2.6	表达式的类型转换	71
6.3	语句	72
	语句是程序的最小单位，程序由一条条语句组成，本节将讲解几条在Java中特殊的语句。	
6.3.1	语句中的空格	73
6.3.2	空语句	73
6.3.3	声明语句	73
6.3.4	赋值语句	74
6.4	练一练	74
6.5	跟我上机	74

第7章 改变程序执行方向——程序控制结构 75

 视频教学录像：35分钟

灵活使用程序控制语句是一个成功Java开发人员的必备技能，本章将循序渐进讲解在Java中程序控制的方法。

7.1	程序的结构设计	76
7.1.1	顺序结构	76
7.1.2	选择结构	76
7.1.3	循环结构	78
7.2	选择结构	78
7.2.1	if语句	78
7.2.2	if...else语句	79
7.2.3	if...else if...else语句	80

7.2.4	条件运算符	82
7.2.5	多重选择——switch语句	83
7.3	循环结构	85
	合理使用循环结构将大大减轻程序工作量，并使程序代码简洁易懂。	
7.3.1	while循环	86
7.3.2	do...while循环	87
7.3.3	for循环	89
7.3.4	循环嵌套	91
7.4	循环的跳转	92
7.4.1	break语句	92
7.4.2	continue语句	93
7.5	练一练	95
7.6	跟我上机	95

第8章 常用的数据结构——数组 96



视频教学录像：1小时2分钟

数组可以模拟生活中的很多模型，比如排序、队列问题等，使用数组可以使程序的编写更科学合理。

8.1	一维数组	97
8.1.1	一维数组的声明与内存的分配	97
8.1.2	数组中元素的表示方法	98
8.1.3	数组初值的赋值	100
8.1.4	数组应用范例	101
8.1.5	与数组操作有关的API方法	102
8.2	二维数组	104
	学习的过程应该由简入繁，掌握一维数组后，二维数组可以看做一维数组的衍生应用，多维数组与此类似。	
8.2.1	二维数组的声明与分配内存	104
8.2.2	二维数组元素的引用及访问	105
8.3	多维数组	106
8.4	练一练	107
8.5	跟我上机	108

第2篇 核心技术

掌握了基础知识，你已经跨进了Java的门槛，本篇将带领你更上一层楼，去探索Java的核心世界。

第9章 面向对象设计——类和对象 110




视频教学录像：2小时13分钟

Java是面向对象的编程语言，类和对象是面向对象编程的重要概念。一个人如果不了解类和对象，就不能说会使用Java语言。

9.1	面向对象程序设计的基本概念	111
9.1.1	对象	111
9.1.2	类	111
9.1.3	封装性	112
9.1.4	继承性	112
9.1.5	多态性	113
9.2	类	113
9.2.1	类的声明	114
9.2.2	类的定义	115
9.3	对象	116
	对象的使用让Java程序在处理现实问题时更加人性化，使用对象可以用“进化”的方式衍生出具有更多特性的模型。	
9.3.1	对象的声明	116
9.3.2	对象的使用	117
9.3.3	对象的比较	119
9.3.4	对象数组的使用	121
9.4	类的属性	123
9.4.1	属性的定义	123
9.4.2	属性的使用	123
9.5	类的方法	125
9.5.1	方法的定义	125
9.5.2	方法的使用	126
9.5.3	构造方法	126
9.5.4	构造方法的重载	128
9.5.5	构造方法的私有	131
9.5.6	在类内部调用方法	134
9.6	练一练	136
9.7	跟我上机	136

第10章 类的封装、继承与多态 137

 视频教学录像：1小时38分钟

封装、继承与多态是类的高级应用，使用这些特性可以使Java程序更加“面向对象”。

10.1	类的封装	138
10.1.1	封装的基本概念	138
10.1.2	类的封装实例	139
10.2	类的继承	144

继承是面向对象语言的必备功能，而且是面向对象的另一个重要特性——多态的基础，所

以理解继承以及如何实现继承相当重要。

10.2.1	继承的基本概念	144
10.2.2	类的继承实例	145
10.3	类的继承专题研究	147
10.3.1	子类对象的实例化过程	147
10.3.2	super关键字的使用	149
10.3.3	限制子类的访问	152
10.3.4	覆写	153
10.4	类的多态	156
10.4.1	多态的基本概念	156
10.4.2	类的多态实例	158
10.5	练一练	160
10.6	跟我上机	160

第11章 抽象类与接口 161



视频教学录像：1小时27分钟

抽象类可以理解为“模板”，在Java中设计者可以使用抽象类的格式创建新的类。

11.1	抽象类的基本概念	162
11.2	抽象类实例	162
11.3	接口的基本概念	166
11.4	接口实例	167
	接口与抽象类十分相似，但又有所不同，本节将以实例讲解接口的具体使用。	
11.5	练一练	170
11.6	跟我上机	170

第12章 关于类的专题研究 171



视频教学录像：3小时26分钟


关于Java的类，每个Java编程人员都需要花很多时间领悟。本章通过对类的专题研究，总结出众多编程人员的宝贵经验，使你快速步入高手行列。

12.1	众类鼻祖——Object类	172
12.2	内部类	174
12.2.1	在类外部引用内部类	178
12.2.2	在方法中定义内部类	179
12.3	匿名内部类	182
12.4	匿名对象	185
12.5	再谈方法	186

方法可以简化程序的结构，把具有特定功能的程序代码独立起来，节省编写相同代码的时


	间，使程序模块化。	
12.5.1	方法的参数与返回值	188
12.5.2	方法的重载	190
12.5.3	将数组传递到方法里	191
12.6	引用数据类型的传递	194
12.7	覆写Object类中的equals方法	197
12.8	接口对象的实例化	200
12.9	this关键字的使用	203
12.10	static关键字的使用	207
12.10.1	静态变量	207
12.10.2	静态方法	211
12.10.3	理解main()方法	213
12.10.4	静态代码块	214
12.11	final关键字的使用	216
12.12	instanceof关键字的使用	217
12.13	练一练	219
12.14	跟我上机	219

第13章 储存类的仓库——Java常用类库 220

 视频教学录像：5小时6分钟


Java类库是JDK中提供的已实现的标准类的集合，使用Java类库可以完成涉及字符串处理、图形、网络等方面的操作。

13.1	API概念	221
13.2	String类和StringBuffer类	221
13.3	基本数据类型的包装类	222
13.4	System类与Runtime类	223
13.4.1	System类	223
13.4.2	Runtime类	224
13.5	Date与Calendar、DateFormat类	225
13.6	Math与Random类	228
13.7	hashCode()方法	228
13.8	对象克隆	230
	“对象克隆”就是把现存对象重新复制一份，应该怎样使用克隆技术呢，本节将为你具体讲解。	
13.9	练一练	232
13.10	跟我上机	232

第14章 包及访问权限 ----- 233
 视频教学录像：43分钟


包是类的一种特殊性质，在管理大型项目时一定要使用到包。利用包可以合理地管理大量的类文件，还可以设置他人对类成员的访问权等。本章将详细讲解包及访问权限的使用。

14.1	包的概念及使用	234
14.1.1	包（package）的基本概念	234
14.1.2	import语句的使用	235
14.1.3	JDK中常见的包	237
14.2	类成员的访问控制权限	237
14.3	Java的命名习惯	240
14.4	打包工具——Jar命令的使用	240
	Jar文件是一种压缩文件，习惯称为“Jar包”，如果开发了许多类，提供给用户时一般会将类压缩到一个Jar文件中。本节将讲解打包工具——Jar命令的使用方法。	
14.5	练一练	241
14.6	跟我上机	241

第15章 异常处理 ----- 242
 视频教学录像：43分钟

程序出错不可避免，Java提供了强大的异常处理机制，所有的异常都被封装到一个类中，在程序出错时会异常抛出。

15.1	异常的基本概念	243
15.1.1	为何需要异常处理	243
15.1.2	简单的异常范例	243
15.1.3	异常的处理	244
15.1.4	异常处理机制的回顾	247
15.2	异常类的继承架构	248
15.3	抛出异常	249
15.3.1	在程序中抛出异常	249
15.3.2	指定方法抛出异常	250
15.4	编写自己的异常类	251
	面对各种各样的异常，Java可以通过继承的方式编写自己的异常类。本节教你在Java中如何编写自己的异常类。	
15.5	练一练	253
15.6	跟我上机	253

第16章 Java类集框架 ----- 254
 视频教学录像：1小时49分钟

Java类集框架可以使程序在处理对象时的方法更加标准化，类集接口是构造类集框架的基础。

16.1	类集接口	255
16.1.1	类集接口	256
16.1.2	List接口	257
16.1.3	集合接口	258
16.1.4	SortedSet接口	258
16.2	Collection接口	258
16.2.1	ArrayList类	259
16.2.2	LinkedList类	262
16.2.3	HashSet类	264
16.2.4	TreeSet类	265
16.3	通过迭代方法访问类集	266
16.4	处理映射	269
	Java 2中增加了映射，映射是一个储存关键字和值的关联，或者说是给定一个关键字，可以得到它的值。	
16.4.1	映射接口	269
16.4.2	映射类	271
16.4.3	比较方法	274
16.5	从以前版本遗留下来的类和接口	277
16.5.1	Enumeration接口	278
16.5.2	Vector类	278
16.5.3	Stack类	281
16.5.4	Dictionary类	282
16.5.5	Hashtable类	283
16.5.6	Properties类	285
16.5.7	Properties类中使用store()和load()方法	287
16.6	练一练	288
16.7	跟我上机	288

第17章 JDK 1.5以上版本的新功能——枚举 289



视频教学录像：49分钟

枚举是被命名的整型常数的集合，枚举在生活中具有很大的实际意义，比如枚举一星期的Sunday、Monday和Tuesday等。

17.1	枚举简介	290
17.2	枚举的作用	290
17.3	枚举的用法	292
17.3.1	常见的枚举定义方法	292
17.3.2	在程序中使用枚举	293
17.3.3	在switch语句中使用枚举	294

17.4	枚举类和枚举关键字	295
17.4.1	枚举类	295
17.4.2	枚举关键字	297
17.4.3	枚举类与枚举关键字的区别	297
17.5	类集对于枚举的支持	298
17.5.1	EnumMap	298
17.5.2	EnumSet	299
17.6	深入了解枚举	301
	枚举的作用在Java中，甚至在所有的计算机语言中，都占有举足轻重的地位。了解枚举，不能够浅尝辄止。本节将带领你一起将枚举熟练化，提高工程水平及工程逻辑度。	
17.6.1	枚举的构造方法	301
17.6.2	枚举的接口	302
17.6.3	在枚举中定义抽象方法	303
17.7	练一练	304
17.8	跟我上机	305
第18章	给编译器看的注释——Annotation	306



视频教学录像：1小时6分钟


Annotation是建立在反射机制之上的功能，通过Annotation可以方便地对程序进行注释操作。

18.1	Annotation	307
18.2	系统内建的Annotation	307
18.2.1	@Override	307
18.2.2	@Deprecated	308
18.2.3	@SuppressWarnings	308
18.3	自定义Annotation	309
18.4	Retention和RetentionPolicy	311
18.5	反射与Annotation	311
18.5.1	取得全部的Annotation	312
18.5.2	加入自定义的Annotation	312
18.6	深入Annotation	314
	Annotation要起作用，必须要依靠反射机制，通过反射可以取得在一个方法上声明的Annotation的全部内容。	
18.6.1	Target	314
18.6.2	Documented注释	315
18.6.3	Inherited	316
18.7	练一练	316

第3篇 高级应用

学习了核心技术后，本篇将带领你去掌握 Java 的高级应用技术，迈进高级开发人员行列。


第19章 齐头并进完成任务——多线程 318

 视频教学录像：2小时

多线程机制可以使计算机资源得到更充分的利用，可以让程序在同一时间内完成很多任务。

19.1	进程与线程	319
19.2	认识线程	319
19.2.1	通过继承Thread类实现多线程	321
19.2.2	通过实现Runnable接口实现多线程	322
19.2.3	两种多线程实现机制的比较	324
19.3	线程的状态	328
19.4	线程操作的一些方法	329
19.4.1	取得和设置线程的名称	330
19.4.2	判断线程是否启动	332
19.4.3	后台线程与setDaemon()方法	334
19.4.4	线程的强制运行	335
19.4.5	线程的休眠	337
19.4.6	线程的中断	338
19.5	多线程的同步	340
	使用线程就一定要考虑到多线程的同步问题，因为如果线程不同步，将会引发很多意想不到的后果，本节将讲解多线程的同步方法。	
19.5.1	同步问题的引出	341
19.5.2	同步代码块	342
19.5.3	同步方法	343
19.5.4	死锁	344
19.6	线程间通信	347
19.6.1	问题的引出	347
19.6.2	问题如何解决	347
19.7	线程生命周期的控制	355
19.8	练一练	357
19.9	跟我上机	357

第20章 文件IO操作 358

 视频教学录像：4小时40分钟

程序运行的数据要保存到文件中，就一定要用到I/O输入输出技术。Java提供的I/O操作能把数据保存到

多种类型的文件中。

20.1	File类	359
20.2	RandomAccessFile类	361
20.3	流类	363
20.3.1	字节流	364
20.3.2	字符流	368
20.3.3	管道流	372
20.3.4	ByteArrayInputStream与ByteArrayOutputStream	375
20.3.5	System.in和System.out	376
20.3.6	打印流	376
20.3.7	DataInputStream与DataOutputStream	378
20.3.8	合并流	382
20.3.9	字节流与字符流的转换	384
20.3.10	IO包中的类层次关系图	387
20.4	字符编码	388
20.5	对象序列化	392
	对象序列化是指把对象转换为数据流的一种实现手段，是文件操作的一个重要概念，通过将对象序列化，可以方便地实现对象的传输及保存。	
20.6	练一练	394
20.7	跟我上机	394

第21章 Java网页小程序——Java Applet 395



视频教学录像：7分钟

Java Applet是经过编译的Java程序，能够在所有支持Java的浏览器中运行。Java Applet跨平台、操作系统，具有广泛的使用。

21.1	Applet程序简介	396
21.2	Applet程序中使用的几个基本方法	397
21.3	在HTML中嵌入Applet程序	399
21.3.1	HTML代码的基本结构	399
21.3.2	Applet标记	400
21.3.3	在HTML中传递Applet程序使用的参数	401
21.4	练一练	403
21.5	跟我上机	403

第22章 Java 网络程序设计 404




视频教学录像：39分钟

网络程序设计是Java程序设计的一个重要应用，使用Java可以轻松地开发出各种类型的网络程序。

22.1	Socket介绍	405
22.2	Socket程序	405
22.3	DatagramSocket程序	413
22.4	网络编程的基本概念	416
22.5	TCP程序实现	416
TCP/IP协议是广泛使用的网络协议，本节将指导你利用Java开发出一个简单的TCP程序实现通讯功能。		
22.5.1	简单的TCP程序	417
22.5.2	Echo程序	418
22.5.3	加入多线程	420
22.6	UDP程序实现	421
22.7	练一练	422
22.8	跟我上机	422

第23章 Java数据库编程 423

 视频教学录像：1小时36分钟

在已有的Java函数库中，有一组专门处理数据库连接的API：JDBC。本章将带领大家学会如何使用来自Java的数据精华——JDBC。

23.1	数据库连接的基本概念	424
23.2	使用数据库的准备工作	425
23.2.1	Oracle数据库的安装	425
23.2.2	数据库连接驱动程序设置	427
23.2.3	数据库表的准备	428
23.3	连接数据库的步骤	430
23.4	数据库连接的详细步骤	430
23.5	数据维护	432
23.5.1	增加数据	432
23.5.2	更新数据	433
23.5.3	删除数据	435
23.6	查询数据库中的内容	437
23.7	查询信息实例	439
23.8	与数据库相关的接口	442
23.8.1	完成增加操作	442
23.8.2	完成查询操作	444
23.8.3	完成模糊查询操作	446
23.9	批处理	447
23.10	事务处理	449
23.11	MySQL数据库	451

MySQL是免费的数据库软件，最新版本的MySQL功能已经十分强大，掌握MySQL对于进

行在Java下的数据库程序开发会有很大帮助。

23.11.1	MySQL数据库的安装	451
23.11.2	MySQL数据库的基本命令	453
23.11.3	使用MySQL数据库	454
23.12	练一练	455
23.13	跟我上机	455

第4篇 项目实战

纸上谈兵终觉浅，在系统学习过Java的各项技术后，本篇将通过2个项目实战教你灵活掌握Java开发大型实用项目。

第24章 Java项目开发实战——五子棋游戏 458



视频教学录像：4小时39分钟

本章将带领读者使用Java语言，从无到有设计出一款有趣好玩的五子棋游戏，所用到的知识包括但不限于之前章节讲到的内容。希望大家一定要亲自动手打造这款游戏，体验编程的乐趣。

24.1	系统概述	459
24.1.1	运行本系统	459
24.1.2	本系统的开发步骤	459
24.1.3	五子棋游戏的功能	460
24.1.4	主要技术	460
24.2	开发前的知识准备之一——Swing编程	460
24.2.1	与窗体相关的类——JFrame	461
24.2.2	与对话框相关的类——JOptionPane	466
24.2.3	与监听鼠标相关的类——MouseListener	469
24.2.4	确定鼠标坐标的类——MouseEvent	471
24.3	开发前的知识准备之二——显示图片的类ImageIO	472
24.4	开发前的知识准备之三——图形的绘制类Graphics	472
24.5	游戏界面开发	472
24.6	绘制棋子	474
24.7	保存棋局	474
24.8	判断游戏胜负	474
24.9	处理屏幕闪烁问题	475
24.10	实现各个功能按钮	475
24.11	完整代码	475
24.11.1	导入部分	475
24.11.2	属性设置	476
24.11.3	主类的构造函数	477
24.11.4	Paint方法	477

24.11.5	监控鼠标	479
24.11.6	判断胜负	483
24.11.7	判断有几个棋子已经连接起来	485

第25章 Java项目开发实战——人事管理 487



视频教学录像：1小时8分钟

通过前面章节的学习，相信读者已经对在Java中开发应用程序的过程比较熟悉了，本章将通过一个人事管理系统的设计，深入学习Java的实际应用项目开发。

25.1	系统概述	488
25.1.1	运行系统	488
25.1.2	系统的开发步骤	491
25.2	系统需求分析	493
25.3	综合描述	493
25.3.1	关键技术	493
25.3.2	名词解释	493
25.3.3	运行环境	493
25.4	概要设计	494
25.4.1	数据库设计	494
25.4.2	接口设计	494
25.4.3	代理	496
25.5	代码实现	497
25.5.1	Person.java	497
25.5.2	IPersonDAO.java	498
25.5.3	DatabaseConnection.java	499
25.5.4	IPersonDAOProxy.java	501
25.5.5	IPersonDAOImpl.java	503
25.5.6	DAOFactory.java	506
25.5.7	Menu.java	507
25.5.8	InputData.java	508
25.5.9	PersonOperate.java	509
25.5.10	Test.java	512

第5篇 王牌资源

实用、专业，这就是王牌。压箱底王牌倾情放送。

王牌资源一览 514

王牌1	Java SE类库查询手册（光盘）	514
-----	-------------------	-----

第 1 篇

基础知识

本篇从 Java 的基本概念及特性开始 Java 的学习之旅，并通过一些实例来了解 Java 的实际开发过程。

读者在学完本篇后将会了解到 Java 软件和编程的基本概念，掌握 Java 的基本操作及应用方法，为后面的学习打好基础。

- ▶ 第 1 章 初识庐山真面目——Java 6
- ▶ 第 2 章 再识庐山真面目——简单的 Java 程序
- ▶ 第 3 章 最流行的 Java 开发工具——Eclipse
- ▶ 第 4 章 最常用的编程元素——常量与变量
- ▶ 第 5 章 不可不知的数据分类法——数据类型
- ▶ 第 6 章 最重要的编程部件——运算符、表达式与语句
- ▶ 第 7 章 改变程序执行方向——程序控制结构
- ▶ 第 8 章 常用的数据结构——数组

第 1 章

初识庐山真面目——Java 6



本章视频教学录像：1 小时 21 分钟

Java语言是一款优秀的编程语言，它的优点是与平台无关，可以实现“一次编写，到处运行”。Java是一款面向对象的语言，它简洁有效，具有高度的可移植性。Java虚拟机（JVM）使经过编译的Java代码在任何系统上都能运行。本章介绍Java的历史、现状和特点，以及如何配置JDK开发环境和编写第1个Java程序。

本章要点（已掌握的在方框中打勾）

- ☐ 了解 Java 的历史与现状
- ☐ 了解 Java 的特点
- ☐ 掌握 Java 开发工具箱 JDK 的安装
- ☐ 掌握开发环境的配置
- ☐ 学会编写第 1 个 Java 程序



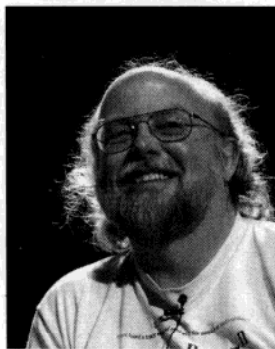
1.1 Java 的历史

▶ 本节视频教学录像：22 分钟

Java 来自于 Sun 公司的一个叫 Green 的项目，其原先的目的是为家用电子消费产品开发一个分布式代码系统，这样就可以把 E-mail 发给电冰箱、电视机等家用电器，对它们进行控制，和它们进行信息交流。开始他们准备采用 C++，但 C++ 太复杂，安全性差，最后基于 C++ 开发了一种新语言 Oak（Java 的前身）。Oak 是一种用于网络的精巧而安全的语言，Sun 公司曾以此投标一个交互式电视项目，但结果被 SGI 打败。于是 Oak 几乎无家可归，恰巧这时 Mark Ardreessen 开发的 Mosaic 和 Netscape 启发了 Oak 项目组成员，他们用 Java 编制了 HotJava 浏览器，得到了 Sun 公司首席执行官 Scott McNealy 的支持，触发了 Java 进军 Internet。



Java 标志



Java 主要设计者 James Gosling

Java 技术是由美国 Sun 公司倡导和推出的，它包括 Java 语言和 Java Media APIs、Security APIs、Management APIs、Java Applet、Java RMI、Java Bean、Java OS、Java Servlet、Java Server Page 以及 JDBC 等。现把 Java 技术的发展重要历程简述如下。

- 1990 年，Sun 公司 James Gosling 领导的小组设计了一种平台独立的语言 Oak，主要用于为各种家用电器编写程序。

- 1995 年 1 月，Oak 被改名为 Java；1995 年 5 月 23 日，Sun 公司在 Sun World '95 上正式发布 Java 和 HotJava 浏览器。

- 1996 年 2 月，Sun 公司发布 Java 芯片系列，包括 PicoJava、MicroJava 和 UltraJava，并推出 Java 数据库连接 JDBC（Java Database Connectivity）。

- 1996 年 4 月，Microsoft 公司、SCO 公司、苹果电脑公司（Apple）、NEC 公司等获得 Java 许可证。Sun 公司宣布允许苹果电脑、HP、日立、IBM、Microsoft、Novell、SGI、SCO、Tandem 等公司将 Java 平台嵌入到其操作系统中。

- 1996 年 6 月，Sun 公司发布 JSP1.0，同时推出 JDK1.3 和 Java Web Server 2.0。

- 1996 年 9 月，Addison-Wesley 和 Sun 公司推出 Java 虚拟机规范和 Java 类库。

- 2000 年 9 月，Sun 公司发布 JSP1.2 和 Java Servlet 2.3 API。

- 2004 年 9 月，J2SE1.5 发布，成为 Java 语言发展史上的又一个里程碑。为了表示该版本的重要性，J2SE1.5 更名为 Java SE 5.0。

- 2005 年 6 月，JavaOne 大会召开，Sun 公司公开 Java SE 6。此时，Java 的各种版本已经更

名，取消了其中的数字“2”：J2EE 更名为 JAVA EE，J2SE 更名为 JAVA SE，J2ME 更名为 JAVA ME。

• 2006 年 12 月，Sun 公司发布 JRE 6.0。

目前 JDK7.0 正在研发中，其测试版在 <https://jdk7.dev.java.net/> 上可以下载使用。

1.2 Java 的现状

▶ 本节视频教学录像：5 分钟

Java 是 Sun 公司推出的新一代面向对象程序设计语言，特别适于 Internet 应用程序开发。但它的平台无关性直接威胁到了 Wintel 的垄断地位，这表现在以下几个方面。

• 计算机产业的许多大公司购买了 Java 许可证，包括 IBM、Apple、DEC、Adobe、SiliconGraphics、HP、Oracle、TOSHIBA 以及 Microsoft 等。这一点说明，Java 已得到了业界的认可。

• 众多的软件开发商开始支持 Java 软件产品。例如 Inprise 公司的 JBuilder、Sun 公司自己做的 Java 开发环境 JDK 与 JRE。Sysbase 公司和 Oracle 公司均已支持 HTML 和 Java。

• Intranet 正在成为企业信息系统最佳的解决方案，而其中 Java 将发挥不可替代的作用。Intranet 的目的是将 Internet 用于企业内部的信息类型，它的优点是便宜、易于使用和管理。用户不管使用何种类型的机器和操作系统，界面是统一的 Internet 浏览器，而数据库、Web 页面、Applet、Servlet、JSP 等则存储在 Web 服务器上，无论是开发人员还是管理人员，或是用户都可以受益于该解决方案。

1.2.1 Java 技术分支

Java 主要有下列 3 类技术分支。

(1) JAVA SE: Java 2 Platform, Standard Edition。

前身为 J2SE，2005 年之后更名为 JAVA SE。

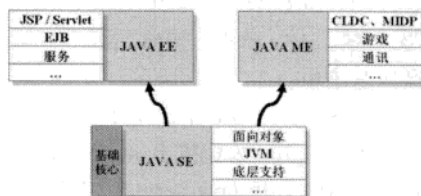
(2) JAVA EE: Java 2 Platform, Enterprise Edition。

前身为 J2EE，2005 年之后更名为 JAVA EE。

(3) JAVA ME: Java 2 Platform, Micro Edition。

前身为 J2ME，2005 年之后更名为 JAVA ME。

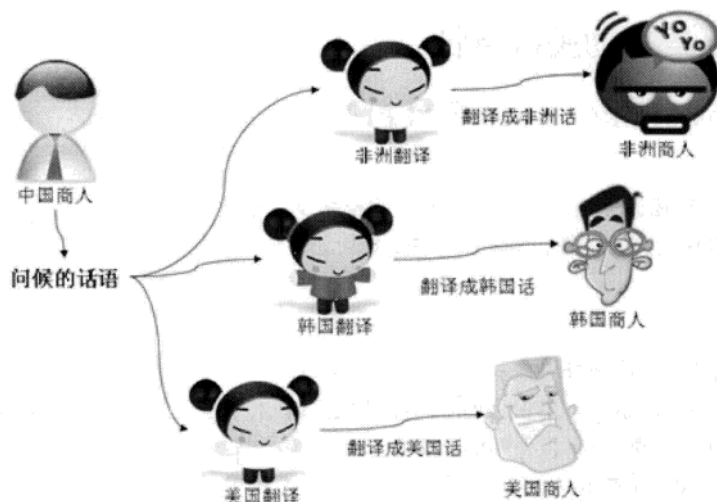
实际上从 Java 的运用来讲，Java 语言现在主要应用在网络上，单机的程序由于微软的问题，造成了发展的中断。JAVA ME 则主要用于完成手机开发。



但是，以上的 3 个程序的分支点，不管如何划分，都是以 JAVA SE 为核心的，所以掌握 JAVA SE 最为重要。

1.2.2 Java 语言的跨平台性

跨平台性就是可以在各个不同的平台间进行程序的移动。比如，一个在 Windows 下开发出来的程序，可以直接在 Linux 下进行运行。所以，在一般的 Java 开发中，所有的开发平台都是在 Windows 下完成的，之后在运行时再部署到 Linux、UNIX 环境之下。



1.3 Java 的特点

▶ 本节视频教学录像：5 分钟

Java 到底是一种什么样的语言？Java 是一种简单的、面向对象的、分布式的、解释型的、健壮的、安全的、结构中立的、可移植的、性能很优异的、多线程的、动态的语言。

1.3.1 Java 语言的优点

Java 语言最大的优点就是与平台无关，在 Windows 9x、Windows NT、Solaris、Linux、MacOS 以及其他平台上，都可以使用相同的代码。“一次编写，到处运行”的特点，使其在互联网上被广泛采用。

由于 Java 语言的设计者十分熟悉 C++ 语言，所以在设计时很好地借鉴了 C++ 语言。可以说，Java 语言是比 C++ 语言“还面向对象”的一种编程语言。Java 语言的语法结构与 C++ 语言的语法结构十分相似，这使得 C++ 程序员学习 Java 语言更加容易。

当然，如果仅仅是对 C++ 改头换面，那么就不会有今天的 Java 热了。Java 语言提供的一些新的特性，使得使用 Java 语言比 C++ 语言更容易写出“无错代码”。

这些新特性有以下几点。

(1) 提供了对内存的自动管理，程序员无需在程序中进行分配、释放内存，那些可怕的内存分配错误不会再打扰设计者了。

(2) 去除了 C++ 语言中的令人费解、容易出错的“指针”，而是用其他的方法来弥补。

(3) 避免了赋值语句（如 `a = 3`）与逻辑运算语句（如 `a == 3`）的混淆。

(4) 取消了多重继承这一复杂的概念。

Java 语言的规范是公开的，可以在 <http://www.sun.com> 上找到它，同时国内的学员可以登录 www.nldnjava.cn（魔乐科技软件学院），下载速度会更快。阅读 Java 语言的规范是提高技术水平的好方法。

1.3.2 Java 语言的关键特性

Java 语言有许多有效的特性，吸引着程序员们，最主要的有以下几个。

1. 简洁有效

Java 语言是一种相当简洁的“面向对象”的程序设计语言。Java 语言省略了 C++ 语言中所有的难以理解、容易混淆的特性，例如头文件、指针、结构、单元、运算符重载、虚拟基础类等。它更加严谨、简洁。

2. 可移植性

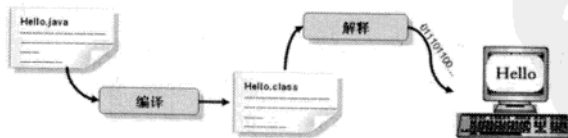
同体系结构无关的特性使得 Java 应用程序可以在配备了 Java 解释器和运行环境的任何计算机系统上运行，这成为 Java 应用软件便于移植的良好基础。但仅仅如此还不够，如果基本数据类型设计依赖于具体实现，也将为程序的移植带来很大的不便。例如在 Windows3.1 中整数(Integer)为 16bit，在 Windows95 中整数为 32bit，在 DECAlpha 中整数为 64bit，在 Intel486 中整数为 32bit。通过定义独立于平台的基本数据类型及其运算，Java 数据得以在任何的硬件平台上保持一致。

3. 面向对象

面向对象可以说是 Java 最重要的特性。Java 语言的设计完全是面向对象的，它不支持类似 C 语言那样的面向过程的程序设计技术。Java 支持静态和动态风格的代码继承及重用。单从面向对象的特性来看，Java 类似于 SmallTalk，但其他特性，尤其是适用于分布式计算环境的特性，则远远超越了 SmallTalk。

4. 解释型

Java 语言是一种解释型语言，相对于 C/C++ 语言来说，用 Java 语言写出来的程序效率低，执行速度慢。但它正是通过在不同的平台上运行 Java 解释器，对 Java 代码进行解释，来实现“一次编写，到处运行”的宏伟目标的。为了达到目标，牺牲效率还是值得的，况且，现在的计算机技术日新月异，运算速度也越来越快，用户是不会感到太慢的。



5. 适合分布式计算

Java 语言具有强大的、易于使用的联网能力，非常适合开发分布式计算的程序。Java 应用程序可以像访问本地文件系统那样通过 URL 访问远程对象。

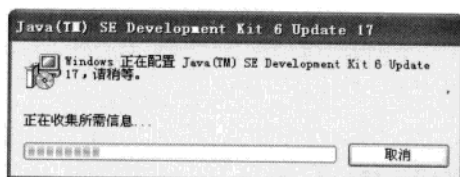
使用 Java 语言编写 Socket 通信程序十分简单，使用它比使用任何其他语言都简单。而且它还十分适用于公共网关接口（CGI）脚本的开发，另外还可以使用 Java 小应用程序（Applet）、Java 服务器页面（Java Server Page，缩写 JSP）和 Servlet 等手段来构建更丰富的网页。

1.4 安装 Java 开发工具箱——JDK 1.6.0_17 多国语言版

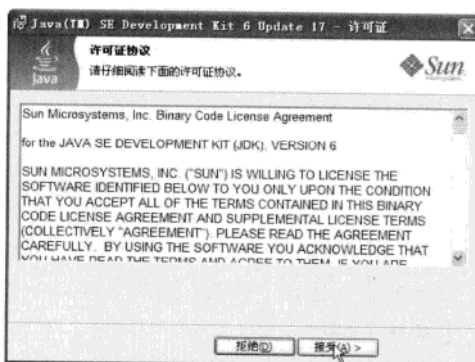
本节视频教学录像：10 分钟

本节介绍 JDK 的安装过程。在这里选用的是 JDK1.6.0_17 版本。具体安装过程如下。

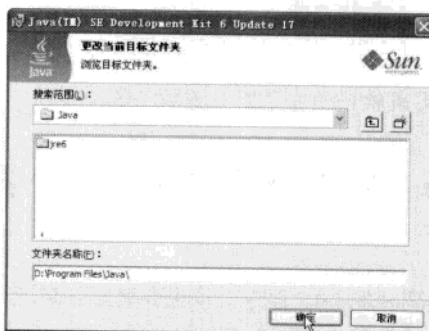
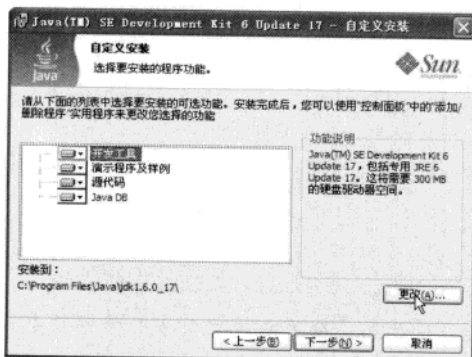
- ① 首先准备好 JDK 的安装文件：
jdk-6u17-windows-i586.exe。
- ② 单击安装文件，安装文件开始解压缩准备安装。
- ③ 解压缩后，系统会弹出【许可证协议】对话框，单击【接受】按钮继续安装。
- ④ 安装系统进入【自定义安装】对话框，单击【更改】按钮，可以修改软件的安装路径。
- ⑤ 进入【更改当前目标文件夹】对话框，在此可以设置和修改 Java 的安装路径，然后单击【确定】按钮，完成目标文件夹的设置。
- ⑥ 返回【自定义安装】对话框，单击【下一步】按钮，继续进行软件的安装。
- ⑦ 软件进入正式安装状态。



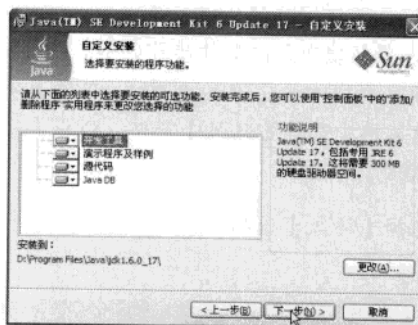
- ③ 解压缩后，系统会弹出【许可证协议】对话框，单击【接受】按钮继续安装。



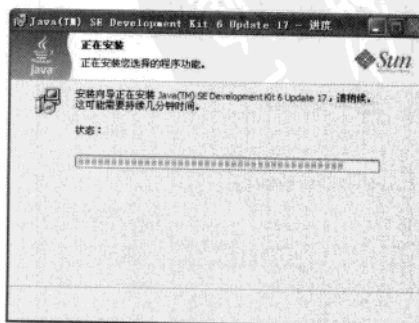
- ④ 安装系统进入【自定义安装】对话框，单击【更改】按钮，可以修改软件的安装路径。



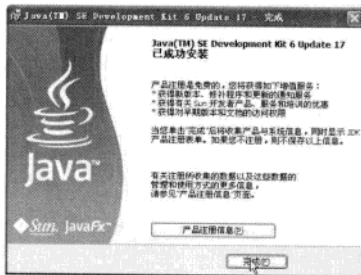
- ⑥ 返回【自定义安装】对话框，单击【下一步】按钮，继续进行软件的安装。



- ⑦ 软件进入正式安装状态。



- ⑧ 软件安装完成，在出现的【完成】对话框中单击【完成】按钮，便完成了该软件的安装。



器进入其官方网站，从中可以完成用户信息的注册和内容的浏览，并可获取更多的技术信息与帮助。



- ⑨ 完成软件的安装后，系统会引导 IE 浏览

1.5 磨刀不误砍柴工——配置开发环境

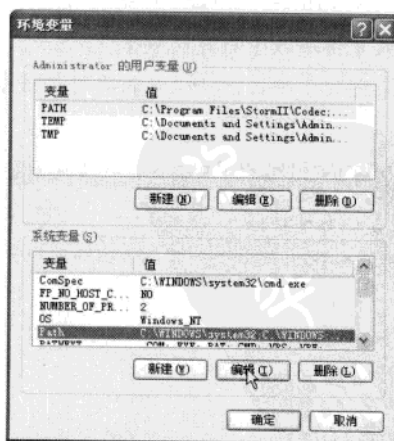
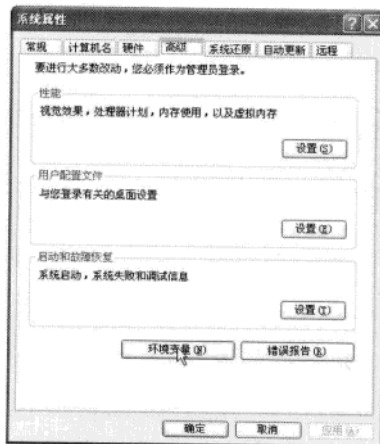
本节视频教学录像：6 分钟

要开发 Java 程序，首先要配置好环境变量，但 Java 的运行环境的配置比较麻烦，相信有些读者也会有这种体会。

在编译 Java 程序时需要用到 javac 这个命令，执行 Java 程序需要 java 这个命令，而这两个命令并不是 Windows 自带的命令，所以使用它们的时候需要配置好环境变量，这样就可以在任何的目录下使用这两个命令了。

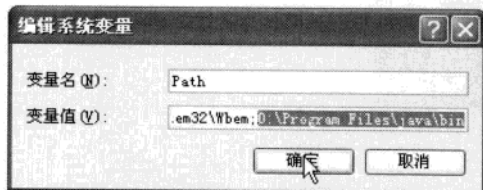
设置环境变量的具体步骤如下。

- ① 在桌面中右击【我的电脑】，在弹出的快捷菜单中选择【属性】命令，弹出【系统属性】对话框。
- ② 选择【高级】选项卡，单击【环境变量】按钮。
- ③ 在弹出的【环境变量】对话框【系统变量】下的列表中选择【Path】选项，然后单击【编辑】按钮。

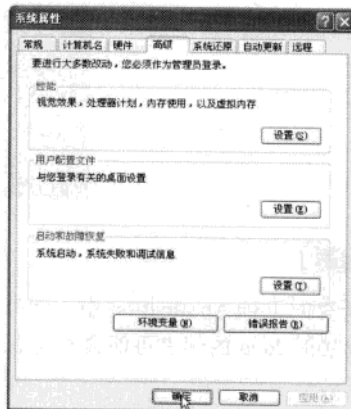


- ④ 在弹出的【编辑系统变量】对话框的【变量值】文本框中输入 JDK 的安装路径，输

入“D:\Program Files\java\bin;”，然后单击【确定】按钮，返回【系统属性】对话框。



- ⑤ 在【系统属性】对话框中单击【确定】按钮，完成环境变量的设置。



提示：“D:\Program Files\java”是安装 JDK 的路径。

在这里使用的是 Windows XP 操作系统。至于其他的操作系统，如 Windows 2000，在设置环境变量的时候与 Windows XP 的配置有许多不同，如果读者感兴趣，可以查阅其他的资料。

1.6 享受安装成果——编写第 1 个 Java 程序

本节视频教学录像：20 分钟

现在就自己来动手编写一个 Java 的程序，亲自感受一下 Java 语言的基本形式。需要说明的是，Java 程序分为两种形式：一种是网页上使用的 Applet 程序（Java 小程序）；另一种是 Application 程序（Java 应用程序）。本书主要讲解的是 Java Application 程序。

【范例 1-1】 编写 Hello.java 程序（代码 1-1.txt）。

```
01 public class Hello
02 {
03     // 是程序的起点，所有程序由此开始运行
04     public static void main(String args[])
05     {
06         // 此语句表示向屏幕上打印“Hello World!”字符串
07         System.out.println("Hello World!");
08     }
09 }
```

【运行结果】

将上面的程序保存为“Hello.java”文件，并在命令行中输入“javac Hello.java”，没有错误后输入“java Hello”。运行结果如图所示，显示“Hello World!”。



【范例分析】

在所有的 Java Application 中，所有的程序都是从 `public static void main(String args[])` 开始运行的。刚接触 Java 的读者可能会觉得有些难记，在后面的章节中会详细讲解 `main` 方法的各个组成部分。

上面的程序如果暂时不明白也没有关系，读者只要将程序一点一点都敲下来，之后按照步骤编译、执行就可以了。在这里只是让读者对 Java Application 程序有一个初步印象，因为以后所有的内容都将围绕 Java Application 程序进行。

1.7 classpath 的指定

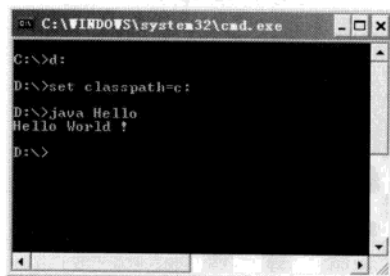
▶ 本节视频教学录像：7 分钟

在 Java 中可以使用 “`set classpath`” 命令指定 java 类的执行路径。下面通过一个例子来了解 `classpath` 的作用，假设这里的 “`Hello.class`” 类位于 “`C:\`” 目录下。

在 “`D:\`” 目录下的命令行窗口执行下面的指令。

```
set classpath=c:
```

之后在 “`D:\`” 目录下执行 “`java Hello`” 命令，如图所示。



从输出结果可以发现，虽然在 “`D:\`” 目录中并没有 “`Hello.class`” 文件，但是却也可以用 “`java Hello`” 执行 “`Hello.class`” 文件。之所以会有这种结果，就是因为 “`D:\`” 目录中使用了 “`set classpath`” 命令，将类的查找路径指向了 “`C:\`” 目录，所以在运行时，会从 “`C:\`” 目录开始查找。



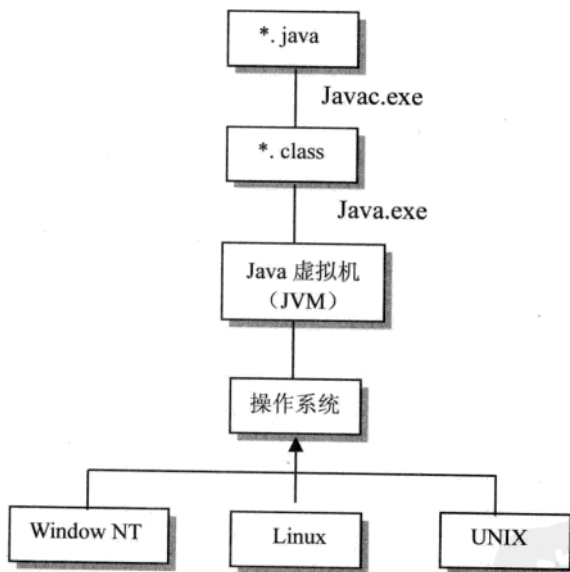
提示：可能有些读者在按照上述的方法操作时，发现并不好用。这里要告诉读者的是，在设置 `classpath` 时，最好将 `classpath` 指向当前目录，即所有的 `class` 文件都从当前文件夹中开始查找：“`set classpath=.`”。

1.8 探秘 Java 虚拟机(JVM)

本节视频教学录像：6 分钟

在 Java 中引入了虚拟机的概念，即在机器和编译程序之间加入了一层抽象的虚拟的机器。这台虚拟的机器在任何平台上都提供给编译程序一个共同的接口。编译程序只需要面向虚拟机，生成虚拟机能够理解的代码，然后由解释器来将虚拟机代码转换为特定系统的机器码执行。在 Java 中，这种供虚拟机理解的代码叫做字节码 (ByteCode)，它不面向任何特定的处理器，只面向虚拟机。每一种平台的解释器是不同的，但是实现的虚拟机却是相同的。Java 源程序经过编译器编译后变成字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上运行。可以说，Java 虚拟机是 Java 语言的基础，它是 Java 技术的重要组成部分。Java 虚拟机是一个抽象的计算机，和实际的计算机一样，它具有一个指令集，并使用不同的存储区域。它负责执行指令，还要治理数据、内存和寄存器。Java 解释器负责将字节代码翻译成特定机器的机器代码。

Java 虚拟机(JVM)是可运行 Java 代码的假想计算机。只要根据 JVM 规范描述将解释器移植到特定的计算机上，就能保证经过编译的任何 Java 代码能够在该系统上运行，如图所示。



从中不难看出 Java 可以实现可移植性的原因，只要在操作系统上 (Windows NT、Linux、UNIX) 植入 JVM (Java 虚拟机)，Java 程序就具有可移植性，也符合 Sun 公司提出的口号 “Write Once, Run Anywhere” (“一次编写，处处运行”)。

1.9 练一练

一、填空题

1. 运行编译后的 class 文件，需要输入命令_____。
2. JVM 是指_____。

3. Java 程序源文件扩展名为_____。

二、简答题

1. 简述 Java 语言的特点。
2. Java 虚拟机的作用是什么？

1.10 跟我上机

编写一个 Java 程序，运行后在控制台中输出“魔乐科技软件学院 www.mldnjava.cn”。



第 2 章

再识庐山真面目——简单的Java程序



本章视频教学录像：8 分钟

麻雀虽小，五脏俱全。本章的实例虽然简单，却基本涵盖了本篇所讲的内容。可以通过本章来概览Java程序的结构及内部部件。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握 Java 程序的组成
- ☐ 掌握 Java 程序注释的使用
- ☐ 掌握 Java 中的标识符和关键字
- ☐ 了解 Java 中的变量及其设置
- ☐ 了解程序的检测
- ☐ 掌握提高程序可读性的方法



2.1 一个简单的例子

本节视频教学录像：3 分钟

从本章开始，正式学习 Java 语言的程序设计。除了认识程序的架构外，本章还将介绍修饰符、关键字以及一些基本的数据类型。通过简单的范例，让读者了解到检测与提高程序可读性的方法，以培养读者正确的程序编写观念与习惯。

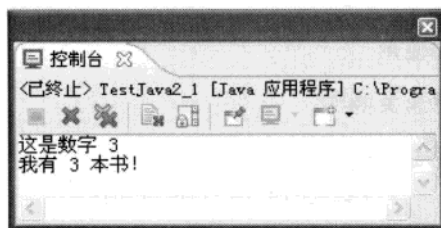
首先来看一个简单的 Java 程序。在介绍程序之前，先简单回顾一下第 1 章讲解的例子，之后再来看下面的这个程序，看看此程序的主要功能是什么。

【范例 2-1】 Java 程序简单范例（代码 2-1.java）。

```
01 // TestJava2_1.java, java 的简单范例
02 public class TestJava2_1
03 {
04     public static void main(String args[])
05     {
06         int num ;           //声明一个整型变量 num
07         num = 3 ;           //将整型变量赋值为 3
08                             //输出字符串，这里用 "+" 号连接变量
09         System.out.println("这是数字 "+num);
10         System.out.println("我有 "+num+" 本书！");
11     }
12 }
```

【运行结果】

保存并运行程序，结果如图所示。



如果现在看不懂上面的这个程序也没有关系，先将它敲进 Java 编辑器里，存盘、编辑、运行，就可以看到输出结果。

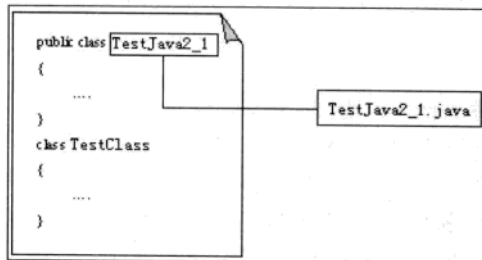
【代码注解】

第 1 行为程序的注释，Java 语言的注释是以“//”标志开始的。注释有助于对程序的阅读与检测，注释在编译时不会被执行。

第 2 行 public class TestJava2_1 中的 public 与 class 是 Java 的关键字，class 为“类”的意

思,后面接上类名称,在本程序中取名为 TestJava2_1。public 则是用来表示该类为公有,也就是在整个程序里都可以访问到它。

需要注意的是:如果将一个类声明成 public,则也要将文件名称取成和这个类一样的名称,如图所示。本例中的文件名为 TestJava2_1.java,而 public 之后所接的类名称也为 TestJava2_1。也就是说,在一个 Java 文件里,最多只能有一个 public 类,否则.java 的文件便无法命名。



第4行 public static void main(String args[]) 为程序运行的起点。第4~10行的功能类似于一般程序语言中的函数(function),但在Java中称之为 method(方法)。因此C语言里的 main() 函数(主函数),在Java中则被称为 main() method(主方法)。

main() method 的主体(body)从第5行的左大括号“{”到第11行的右大括号“}”为止。每一个独立的Java程序一定要有 main() method 才能运行,因为它是程序开始运行的起点。

第6行“int num;”的目的是声明 num 为一个整数类型的变量。在使用变量之前必须先声明其类型。

第7行“num=3;”为一赋值语句,即把整数3赋给存放整数的变量 num。

第9行的语句如下。

```
09      System.out.println("这是数字 "+num);
```

程序运行时会在显示器上输出引号(”)内所包含的内容,包括“这是数字”和整数变量 num 所存放的值两部分内容。

System.out 是指标准输出,通常与计算机的接口设备有关,如打印机、显示器等。其后所续的 println,是由 print 与 line 所组成的,意思是将后面括号中的内容打印在标准输出设备——显示器上。因此第9行的语句执行完后会换行,也就是把光标移到下一行的开头继续输出。读者可以把 System.out.println() 改成 System.out.print(), 看一下换行与不换行的区别。

第11行的右大括号告诉编译器 main() method 到这儿结束。

第12行的右大括号告诉编译器 class TestJava2_1 到这儿结束。

这里只是简单地介绍了一下 TestJava2_1 这个程序,相信读者已经对Java语言有了初步的了解。TestJava2_1 程序虽然很短,却是一个相当完整的Java程序。在后面的章节中,将会对Java语言的细节部分做详细的讨论。

2.2 感性认识 Java 程序

▶ 本节视频教学录像: 5 分钟

本节探讨Java语言的一些基本规则及用法。

2.2.1 认识 Java 程序的框架

1. 大括号、段及主体

将类名称定出之后，就可以开始编写类的内容。左大括号“{”为类的主体开始标记，而整个类的主体至右大括号“}”结束。每个命令语句结束时，必须以分号“;”做结尾。当某个命令的语句不止一行时，必须以一对大括号“{}”将这些语句包括起来，形成一个程序段（segment）或是块（block）。

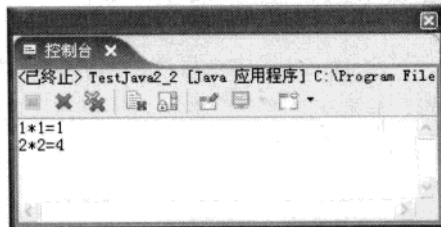
下面以一个简单的程序为例来说明什么是段与主体。若暂时看不懂 TestJava2_2 这个程序，也不用担心，以后会讲到该程序中所用到的命令。在下面的程序中，可以看到 main() method 的主体以左右大括号包围起来；for 循环中的语句不止一行，所以使用左右大括号将属于 for 循环的段内容包围起来；整个程序语句的内容又被第 3 行与第 12 行的左右大括号包围，这个块属于 public 类 TestJava2_2 所有。此外，应注意到每个语句结束时，都是以分号作为结尾。

【范例 2-2】 简单的 Java 程序（代码 2-2.java）。

```
01 //TestJava2_2,简单的 Java 程序
02 public class TestJava2_2
03 {
04     public static void main(String args[])
05     {
06         int x;
07         for(x=1;x<3;x++)
08         {
09             System.out.println(x+"*"+x+"="+x*x);
10         }
11     }
12 }
```

【运行结果】

运行程序并保存，结果如图所示。



2. 程序运行的起始点 —— main() method

Java 程序是由一个或一个以上的类组合而成，程序起始的主体也是被包含在类之中。这个起始的地方称为 main()，用左右大括号将属于 main()段的内容包围起来，称之为 method（方法）。main()方法为程序的主方法，在一个 Java 程序中有且只能有一个 main()方法，它是程序运行

的开端。通常看到的 main() method 如下面的语句片段所示。

```
public static void main(String args[]){    // main() method, 主程序开始
...
}
```

如前一节所述, main() method 之前必须加上 public static void 这 3 个标识符。public 代表 main() 公有的 method; static 表示 main() 在没有创建类对象的情况下, 仍然可以被运行; void 则表示 main() 方法没有返回值。main 后面括号 () 中的参数 String args[] 表示运行该程序时所需要的参数, 这是固定的用法, 如果现在不了解也没有关系, 在以后会一一介绍。

2.2.2 认识标识符

Java 中的包、类、方法、参数和变量的名字, 可由任意顺序的大小写字母、数字、下划线 (_) 和美元符号 (\$) 等组成, 但标识符不能以数字开头, 不能是 Java 中的保留关键字。

下面是合法的标识符。

```
yourname      your_name      _yourname      $yourname
```

下面是非法的标识符。

```
class      67.9      Hello Careers
```



提示：一些刚接触编程语言的读者可能会觉得记住上面的规则很麻烦, 所以在这里提醒读者, 标识符最好永远用字母开头, 而且尽量不要包含其他的符号。

2.2.3 认识关键字

和其他语言一样, Java 中也有许多关键字 (也叫保留字), 如 public、static 等, 这些关键字不能当做标识符使用。下表列出了 Java 中的关键字。这些关键字并不需要读者去强记, 因为在程序开发中一旦使用了这些关键字做标识符, 编辑器会自动提示错误。

abstract	boolean	break	byte	case	catch
char	class	continue	default	do	double
else	extends	false	final	finally	float
for	if	implements	import	instanceof	int
interface	long	native	new	null	package
private	protected	public	return	short	static
synchronized	super	this	throw	transient	true
try	void	volatile	while	const	goto



注意：虽然 goto、const 在 Java 中并没有任何意义, 却也是保留关键字, 与其他的关键字一样, 在程序里不能用来作为自定义的标识符。

2.2.4 认识注释

为程序添加注释可以解释程序的某些语句的作用和功能，提高程序的可读性。也可以使用注释在原程序中插入设计者的个人信息。此外，还可以用程序注释来暂时屏蔽某些程序语句，让编译器暂时不要处理这部分语句，等到需要处理的时候，只需把注释标记取消即可。Java 里的注释根据不同的用途分为以下 3 种类型。

- (1) 单行注释
- (2) 多行注释
- (3) 文档注释

单行注释，就是在注释内容的前面加双斜线 (//)，Java 编译器会忽略这部分信息。如下所示。

```
int num; //定义一个整数
```

多行注释，就是在注释内容的前面以单斜线加一个星形标记 (/*) 开头，并在注释内容末尾以一个星形标记加单斜线 (*/) 结束。当注释内容超过一行时，一般可使用这种方法，如下所示。

```
/*  
int c = 10;  
int x = 5;  
*/
```

文档注释，是以单斜线加两个星形标记 (/**) 开头，并以一个星形标记加单斜线 (*/) 结束。用这种方法注释的内容会被解释成程序的正式文档，并能包含进如 javadoc 之类的工具生成的文档里，用以说明该程序的层次结构及其方法。

2.2.5 认识变量

变量在程序语言中扮演着最基本的角色。变量可以用来存放数据，而使用变量之前则必须先声明它所预保存的数据类型。接下来看看 Java 中变量的使用规则。

1. 变量的声明

举例来说，想在程序中声明一个可以存放整数的变量，这个变量的名称为 num。在程序中即可写出如下所示的语句。

```
int num; //声明 num 为整数变量
```

int 为 Java 的关键字，代表整数 (Integer) 的声明。若要同时声明多个整型的变量，可以像上面的语句一样分别声明它们，也可以把它们都写在同一个语句中，每个变量之间以逗号分开。

2. 变量名称

读者可以依据个人的喜好来决定变量的名称，这些变量的名称不能使用 Java 的关键字。通常会以变量所代表的意义来取名 (如 num 代表数字)。当然也可以使用 a、b、c 等简单的英文字

母代表变量，但是当程序很大时，需要的变量数量会很多，这些简单名称所代表的意义就比较容易忘记，必然会增加阅读及调试程序的困难度。

3. 变量的设置

给所声明的变量赋予一个属于它的值，用等号运算符(=)来实现。具体可使用如下所示的3种方法进行设置。

(1) 在声明变量时设置。

举例来说，在程序中声明一个整数的变量 num，并直接把这个变量赋值为 2，可以在程序中写出如下的语句。

```
int num = 2;           //声明变量，并直接设置
```

(2) 声明后再设置。

一般来说也可以在声明后再给变量赋值。举例来说，在程序中声明整数的变量 num1、num2 及字符变量 ch，并且给它们分别赋值，在程序中即可写出下面的语句。

```
int num1,num2;         //声明变量
char c;
num1 = 2;              //赋值给变量
num2 = 3;
ch = 'z';
```

(3) 在程序的任何位置声明并设置。

以声明一个整数的变量 num 为例，可以等到要使用这个变量时再给它赋值。

```
int num;               //声明变量
...
num = 2;               //用到变量时，再赋值
```

2.2.6 认识数据类型

除了整数类型之外，Java 还提供有多种数据类型。Java 的变量类型可以是整型(int)、长整型(long)、短整型(short)、浮点型(float)、双精度浮点型(double)，或者字符型(char)和字符串型(String)等。关于这些数据类型，将在第5章介绍。

2.2.7 认识运算符和表达式

计算机是用来做计算的。要运算就要使用运算符。最简单的运算符是“+”、“-”、“*”、“/”、“%”。

表达式是由操作数与运算符所组成，操作数可以是常量、变量，也可以是方法。

2.2.8 认识类

Java 程序是由类 (class) 所组成。类的概念在以后会讲解, 读者只要记住所有的 Java 程序都是由类组成的就可以了。下面的程序片段即为定义类的典型范例。

```
public class Test    // 定义 public 类 —— Test
{
    ...
}
```

程序定义了一个新的 public 类 Test, 这个类的原始程序的文件名称应取名为 Test.java。类 Test 的范围由一对大括号所包含。public 是 Java 的关键字, 指的是对于类的访问方式为公有。

需要注意的是: 由于 Java 程序是由类所组成, 因此在完整的 Java 程序里, 至少需要有一个类。此外, 本书曾在前面提到过在 Java 程序中, 其原始程序的文件名不能随意命名, 必须和 public 类名称一样, 因此在一个独立的原始程序里, 只能有一个 public 类, 却可以有許多 non-public 类。

此外, 若是在一个 Java 程序中没有一个类是 public, 那么对该 Java 程序的文件名就可以随意命名了。

2.3 程序的检测

现在相信读者大概可以依照例子写出几个类似的程序了。接下来做一些小检测, 看看读者能否准确地找出下面的程序中存在的错误。

2.3.1 语法错误

通过下面的范例应学会怎样找出程序中的语法错误。

【范例 2-3】 找出下面程序中的语法错误 (代码 2-3.java)。

```
01 //下面程序的错误属于语法错误, 在编译的时候会自动检测到
02 public class TestJava2_3
03 {
04     public static void main(String args[])
05     {
06         int num1 = 2;           //声明整数变量 num1, 并赋值为 2
07         int num2 = 3;           //声明整数变量 num2, 并赋值为 3
08
09         System.out.println("我有 "+num1" 本书! ");
10         System.out.println("你有 "+num2+"本书! ")
```

```

11    )
12 }

```

【范例分析】

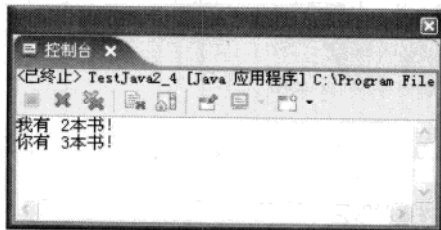
程序 TestJava2_3 在语法上犯了几个错误，若是通过编译器编译，便可以把这些错误找出来。首先，可以看到第 4 行，main() method 的主体以左大括号开始，应以右大括号结束。所有括号的出现都是成双成对的，因此第 11 行 main() method 主体结束时应以右大括号做结尾，而 TestJava2_3 中却以右括号“)”结束。

注释的符号为“//”，但是在第 7 行的注释中，没有加上“//”。在第 9 行，字符串的连接中少了一个“+”号。最后，还可以看到在第 10 行的语句结束时，少了分号作为结束。

上述的 3 个错误均属于语法错误。当编译程序发现程序语法有错误时，会把这些错误的位置指出来，并告诉设计者错误的类型，即可根据编译程序所给予的信息加以更正。将程序更改后重新编译，若还是有错误，再依照上述的方法重复测试，这些错误就会被一一改正，直到没有错误为止。

【运行结果】

保存并运行程序，结果如图所示。



2.3.2 语义错误

若程序本身的语法都没有错误，但是运行后的结果却不符合设计者的要求，此时可能犯了语义错误，也就是程序逻辑上的错误。读者会发现，想要找出语义错误则比找出语法错误更难。

举例来说，想在程序中声明一个可以存放整数的变量，这个变量的名称为 num。

在程序中即可写出如下所示的语句。

【范例 2-4】 程序语义错误的检测（代码 2-4.java）。

```

01 //下面这段程序原本是要计算一共有多少本书，但是由于错把加号写成了减号，
    //所以造成了输出结果不正确，这属于语义错误
02 public class TestJava2_4
03 {
04     public static void main(String args[])
05     {
06         int num1 = 4;           //声明一整型变量 num1
07         int num2 = 5;           //声明一整型变量 num2

```

```

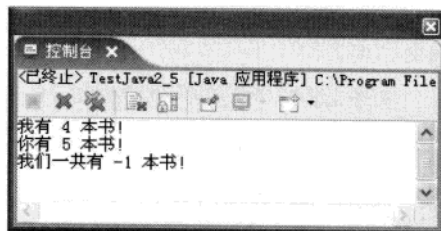
08
09     System.out.println("我有 "+num1+" 本书!");
10     System.out.println("你有 "+num2+" 本书!");
11     //输出 num1-num2 的值 s
12     System.out.println("我们一共有 "+(num1-num2)+" 本书!");
13 }
14 }
    
```

【范例分析】

可以发现，在程序编译过程中并没有发现错误，但是运行后的结果却不正确，这种错误就是语义错误。在第 12 行中，因失误将“num1+num2”写成了“num1-num2”，虽然语法是正确的，但是却不合程序的要求，只要将错误更正后，程序的运行结果就是想要的了。

【运行结果】

保存并运行程序，结果如图所示，显然不符合设计的要求。



2.4 提高程序的可读性

能够写出一个程序的确很让人兴奋，但如果这个程序除了本人之外，其他人都很难读懂，那这就不算是一个好的程序。所以每个程序的设计者在设计程序的时候，也要学习程序设计的规范格式。除了前面所说的加上注释之外，还应当保持适当的缩进，可以看见上面的范例程序都是按缩进的方法编写的，是不是觉得看起来很清晰、明白。读者可以比较下面的两个范例，相信看完之后，就会明白程序中使用缩进的好处了。

【范例 2-5】 缩进格式的程序（代码 2-5.java）。

```

01 //以下这段程序是有缩进的样例，可以发现这样的程序看起来比较清楚
02 public class TestJava2_5
03 {
04     public static void main(String args[])
05     {
06         int x;
07
    
```

```

08      for(x=1;x<=3;x++)
09      {
10          System.out.print("x = "+x+", ");
11          System.out.println("x * x = "+(x*x));
12      }
13  }
14  }

```

【范例 2-6】 非缩进格式的程序（代码 2-6.java）。

```

01  //下面的程序与前面程序的输出结果是一样的，但不同的是，
02  //这个程序没有采用任何缩进，所以看起来很累
03  public class TestJava2_6{
04      public static void main(String args[]){
05          int x ; for(x=1;x<=3;x++){
06              System.out.print("x = "+x+", ");
07              System.out.println("x * x = "+(x*x));}}}

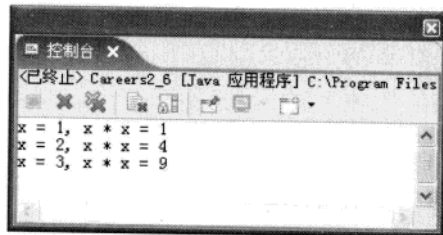
```

【范例分析】

TestJava2_6 这个例子虽然简短，而且语法也没有错误，但是因为编写风格的关系，阅读起来肯定没有 TestJava2_5 这个程序好读，所以建议读者尽量使用缩进，养成良好的编程习惯。

【运行结果】

保存并运行程序，结果如图所示。



2.5 练一练

一、填空题

1. Java 程序是从_____处开始运行的。
2. 在 Java 中，多行注释的开始和结束标记分别为_____和_____。
3. 声明 1 个名称“count”的整型变量的语句为_____。
4. Java 程序中的标识符可由_____、_____和_____组成，但不能以_____开头，不能包含_____。

二、简答题

1. 简述设置变量的 3 种方法。
2. 简述提高程序可读性的两种方法。

2.6 跟我上机

分析下面程序代码的运行结果，并运行程序查看实际结果。

```
01 public class Xiti_2
02 {
03     public static void main (String args[])
04     {
05         int x=10;
06         int y=2;
07         System.out.println(x/y);
08     }
09 }
```



第 3 章

最流行的Java开发工具——Eclipse



本章视频教学录像：38 分钟

工欲善其事，必先利其器。要学习和使用Java语言进行程序开发，就必须选择一种功能强大、使用方便且能够辅助程序开发的IDE集成开发工具，而Eclipse就是目前最为流行的Java语言辅助开发工具。它具有强大的代码辅助功能，能够帮助程序开发人员自动完成输入语法、补全文字、修正代码等操作，能够大量减少程序开发人员的时间和精力。通过本章的学习，读者能够初步了解Eclipse开发工具，并且能够使用它完成程序的开发工作。

本章要点（已掌握的在方框中打勾）

- ☐ Eclipse 概述
- ☐ 掌握 Eclipse 的安装、设置与启动方法
- ☐ 熟悉 Eclipse 的开发环境
- ☐ 使用 Eclipse 创建 Java 程序
- ☐ 在 Eclipse 中调试 Java 程序



3.1 认识 Eclipse 开发工具

本节视频教学录像：9 分钟

本节主要讲解 Eclipse 工具安装、配置以及工作台等内容。

3.1.1 Eclipse 概述

Eclipse 是 IBM 花巨资开发的 IDE 集成开发环境 (Integrated Development Environment)。其前身是 IBM 的 Visual Age for Java(VA4J)。Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台。就其本身而言,它只是一个框架和一组服务,用于通过插件组件构建开发环境是可扩展的体系结构,可以集成不同软件开发供应商开发的产品,将他们开发的工具和组件加入到 Eclipse 平台中。另外 Eclipse 还附带了一个标准的插件集,包括 Java 开发工具 (Java Development Tools, JDT)。

随 Java 应用的日益广泛,各大主要软件供应商都参与到 Eclipse 架构开发中,使得 Eclipse 的插件数量与日增加。Eclipse 为程序开发人员提供了优秀的 Java 程序开发环境。

3.1.2 Eclipse 的安装、设置与启动

Eclipse 的安装非常简单,仅需对下载后的压缩文件进行解压缩即可完成操作。

1. 安装 Eclipse 开发工具

- (1) 可以到官方网站 www.eclipse.org 中下载 3.2.1 版 Eclipse 开发工具。
- (2) 对下载名称为 eclipse-SDK-3.2.1-win32.zip 的 Eclipse 软件进行解压缩。
- (3) 为了便于管理,将解压缩后的 eclipse 文件夹剪切到 D:\Program Files 文件夹中。此时如果运行 D:\Program Files\eclipse\eclipse.exe 可执行文件,便可启动英文版的 Eclipse 开发工具。

2. Eclipse 的国际化

完成 Eclipse 的安装后,从初学者的角度考虑,如果开发工具是中文版的,则更适合我们的语言习惯,在学习和使用的过程中会轻松很多,另外一点就是可以方便地查阅中文教程和帮助信息。为此可以到 Eclipse 网站上下载多国语言包,从而实现 Eclipse 操作界面的国际化,进而用更适合我们学习的中文语言来显示 Eclipse 程序界面。

使 Eclipse 操作界面国际化的具体步骤如下。

- (1) 到官方网站下载与 Eclipse 相匹配的多国语言包。与上文所安装的 eclipse-SDK-3.2.1 版本相匹配的语言包是 NLpack1-eclipse-SDK-3.2.1-win32.zip。



提示:在下载多国语言包时一定要选择和所安装的版本相一致的多国语言包。在语言包的命名名称中也可以识别语言包的类型,如 NLpack1 是亚洲的语言包, NLpack2、NLpack3、NLpack4 则分别是其他洲的语言包。

- (2) 对所下载的 NLpack1-eclipse-SDK-3.2.1-win32.zip 文件进行解压缩。
- (3) 解压缩后包含一个 eclipse 文件夹，该文件夹中包含有 features 和 plugins 两个文件夹。
- (4) 将 features 和 plugins 两个文件夹复制到所安装的 eclipse 根目录中，即 D:\Program Files\eclipse 文件夹中，替换其中对应的文件夹。
- (5) 这样即可实现 Eclipse 的中文语言操作界面。

3. 启动 Eclipse

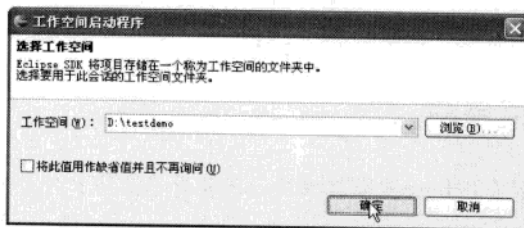
完成了 Eclipse 多国语言包的配置，接下来可以启动 Eclipse。Eclipse 的启动很简单，直接在 Eclipse 的安装文件夹中运行 eclipse.exe 文件即可。具体步骤如下。

- (1) 运行 D:\Program Files\eclipse\eclipse.exe 文件。



注意：为了便于启动，可右击 eclipse.exe 文件，在弹出的快捷菜单中执行【发送到】>【桌面快捷方式】命令，为 Eclipse 启动在桌面创建一个快捷方式。

- (2) Eclipse 启动后，在弹出的【工作空间启动程序】对话框的【工作空间】文本框中输入“D:\testdemo”，然后单击【确定】按钮。

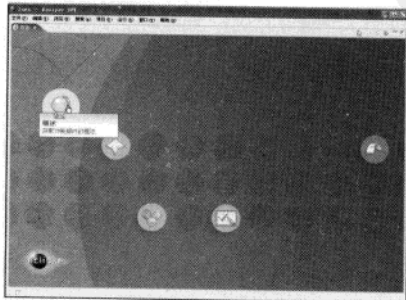


该对话框用于设置 Eclipse 的工作空间，工作空间用于保存 Eclipse 所建立的程序项目和相关的设置。本书所使用的 Eclipse 工作空间为 D:\testdemo。



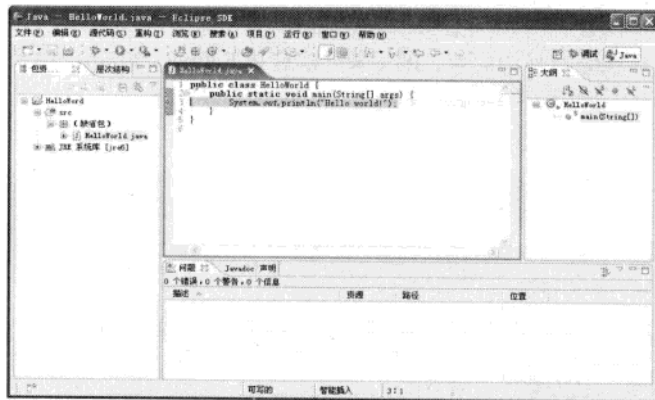
注意：Eclipse 每次启动都会出现设置工作空间的对话框，如果选中【将此值用作缺省值并且不再询问】复选框，就可以将当前的工作空间设置为默认的工作空间，这样再启动 Eclipse 时就不会出现此对话框了。

- (3) 单击【确定】按钮，系统将出现 Eclipse 的欢迎界面，其中包含【概述】、【新增内容】、【样本】、【教程】以及工作台相关按钮和菜单栏等。



3.1.3 Eclipse 工作台

在 Eclipse 的欢迎界面中,单击【工作台】按钮或者关闭【欢迎】的界面窗口,将显示出 Eclipse 的工作台。Eclipse 工作台是程序人员开发结合调试程序的主要场所。另外还可以将其他插件采用无缝衔接的方式集成到该工作台中,当然也可以在该工作台中开发各种插件。



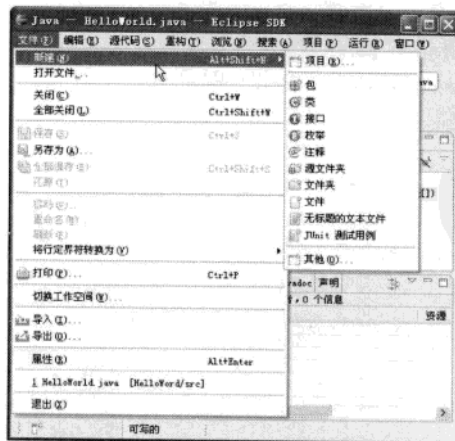
Eclipse 工作台主要有标题栏、菜单栏、工具栏、编辑器、透视图和相关的视图等。

3.1.4 Eclipse 菜单栏

Eclipse 的菜单中包含了 Eclipse 的基本命令,主要有【文件】、【编辑】、【窗口】和【帮助】等菜单。

1. 【文件】菜单

【文件】菜单主要包含【新建】、【保存】、【关闭】以及【刷新】等命令,主要用于新项目的创建、保存以及关闭等操作。



2. 【编辑】菜单

【编辑】菜单主要用于辅助程序代码设计工作,如代码的【复制】、【剪切】和【粘贴】等。



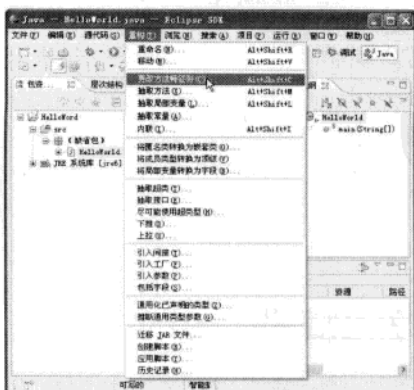
3. 【源代码】菜单

【源代码】菜单中所包含的命令都是和代码编写相关的，主要用于复制编程工作。



4. 【重构】菜单

【重构】菜单是 Eclipse 最为关键的菜单，主要包括对项目重构的相关命令，对本菜单需要重点掌握。



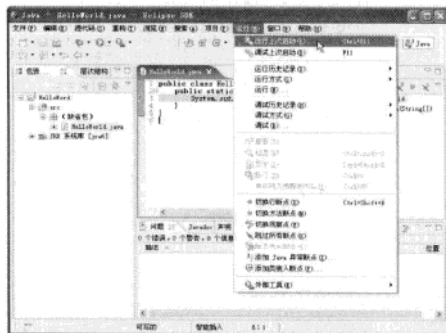
5. 【项目】菜单

【项目】菜单主要用于管理 Eclipse 中的项目，用于项目的打开与关闭、自动构建等操作。



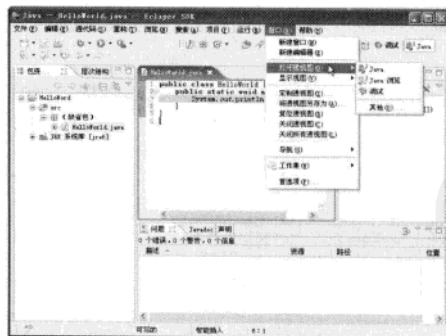
6. 【运行】菜单

【运行】菜单中包含了与程序运行和调试相关的各种操作，同时还具有保存运行和调试的记录功能。



7. 【窗口】菜单

【窗口】菜单主要用于显示、隐藏或处理 Eclipse 中的各种视图和透视图。



3.2 使用 Eclipse 开始工作

本节视频教学录像：8 分钟

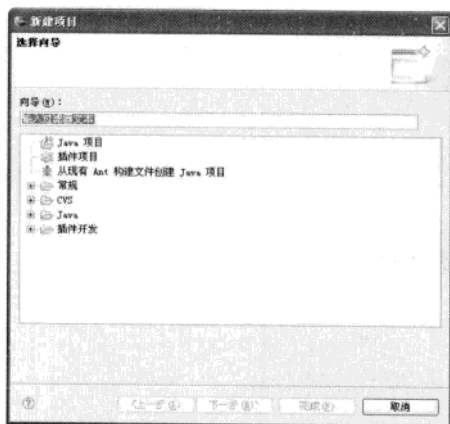
通过前面的学习，读者对 Eclipse 工具应该有了一个基本的认识。本节学习如何使用 Eclipse 完成 Hello Word 程序的编写和运行。

3.2.1 创建 Java 项目

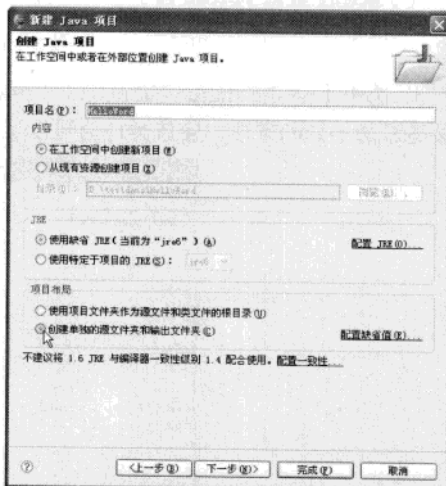
【范例 3-1】 创建 Hello Word 项目。使用项目向导创建一个 Java 项目。

在 Eclipse 中编写应用程序时，需要先创建一个项目。在 Eclipse 中有多种项目，其中 Java 项目是用于管理和编写 Java 程序的。创建 Java 项目的具体步骤如下。

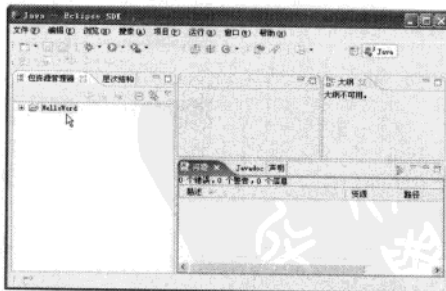
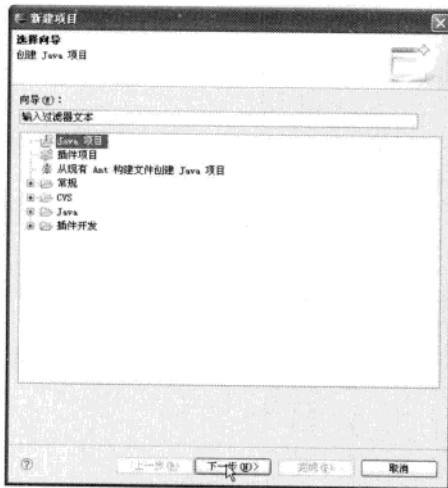
- ① 选择【文件】>【新建】>【项目...】命令，打开【新建项目】对话框。
- ② 在弹出的【新建 Java 项目】对话框的【项目名】文本框中输入【HelloWord】文本，在【项目布局】栏中选中【创建单独的源文件夹和输出文件夹】单选按钮。



- ③ 单击【完成】按钮，完成 Java 项目的创建。



- ④ 单击【完成】按钮，完成 Java 项目的创建。在【包资源管理器】窗口中便会出现一个名称为【HelloWord】的 Java 项目。




3.2.2 创建 Java 类文件

【范例 3-2】 创建 Hello Word 类文件。使用向导创建一个 Java 类文件。

通过前面创建 Java 项目的操作，在工作空间中已经有一个【Java 项目】了。构建 Java 应用

程序的下一个操作就是要创建 HelloWorld 类。创建 Java 类的具体步骤如下。

单击工具栏中的【创建类】按钮  或者在菜单栏中执行【文件】>【新建】>【类】命令，启动【新建 Java 类】向导。

(1) 在【源文件夹】文本框中输入 Java 项目源程序的文件夹位置。通常系统向导会自动填写，如无特殊情况，不需要修改。

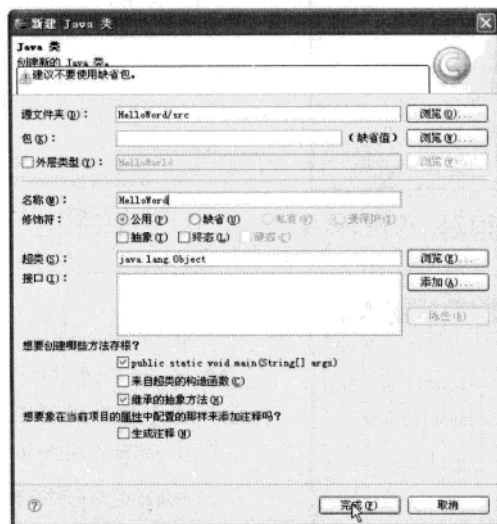
(2) 在【包】文本框中可以输入该 Java 类文件准备使用的包名，系统默认为空，这样会使用 Java 项目的【缺省包】。

(3) 在【名称】文本框中输入新建类的名称，如“HelloWorld”。

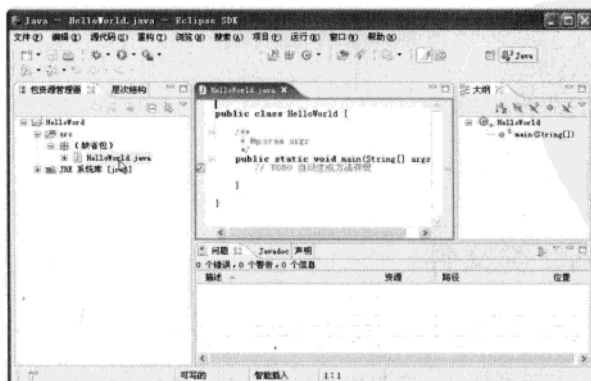


注意：在这里使用的类的名称和项目的名称一致，这并不影响使用，它们分别代表类文件和 Java 项目文件，需要注意区分。

(4) 选中【public static void main (String [] args)】复选框，向导在创建类文件时，会自动为该类添加 main()方法，使该类成为可以运行的主类。



(5) 单击【完成】按钮，完成 Java 类的创建。



3.2.3 在代码编辑器中编写 Java 程序代码

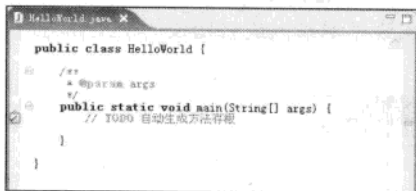
【范例 3-3】 在代码编辑器中编写 Hello Word 程序代码（代码 3-3.java）。

编辑器位于 Eclipse 工作台的中间区域，该区域可以重叠放置多个编辑器。编辑器的类型也可以不同，但是主要的功能都是完成 Java 程序、XML 配置等代码编写或者进行可视化设计工作。下面介绍如何使用该编辑器和代码辅助功能，来快速编写 Java 应用程序。

1. 打开编辑器

当使用创建 Java 类向导完成 Java 类文件的创建后，在 Eclipse 的工作台上会自动打开 Java 编辑器新创建的 Java 类文件。打开 Java 编辑器的方法如下。

- (1) 在【包资源管理器】窗口中，双击或者右击 Java 源文件。
- (2) 在弹出的快捷菜单中执行【打开】命令，便可打开 Java 编辑器界面。



Java 编辑器能以各种样式和不同的颜色来突出显示 Java 语法。其中被突出显示的语法包括程序代码注释、Javadoc 注释、Java 关键字、变量以及字符串等。

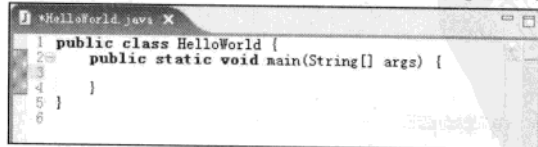


提示：在 Java 代码编辑器的左侧右击，在弹出的快捷菜单中选择【显示行号】菜单项，可以启动 Java 编辑器自动显示行号的功能。

2. 编写 Java 程序代码

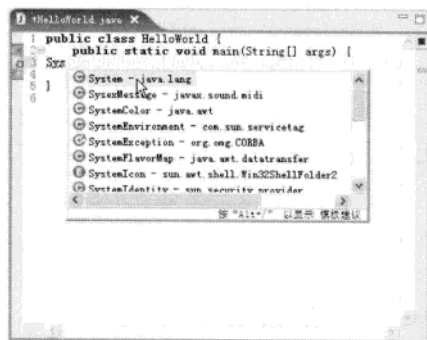
Eclipse 具有强大的突出显示 Java 语法和代码辅助功能。在编写 Java 程序代码时，可以使用【Ctrl+Alt+/】组合键自动补全 Java 关键字，也可以使用【Alt+/】快捷键启动 Eclipse 的代码辅助菜单。下面介绍如何使用 Eclipse 的代码辅助功能完成 Hello world 类的代码编写，具体的操作步骤如下。

- (1) 在【包资源管理器】窗口中，双击【HelloWorld.java】Java 源文件。
- (2) 在 Java 代码编辑器的左侧右击，在弹出的快捷菜单中选择【显示行号】菜单项。



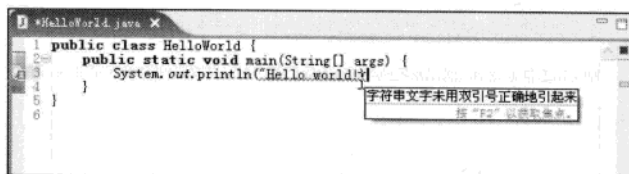
注意：我们已经将代码编辑窗口中无用的内容删除了，下面仅需在第 3 行代码中输入“System.out.println(“Hello world!”);”代码，就可以完成输出 Hello world!语句的功能。

(3) 在第 3 行代码中输入 sys 后按住【Alt + /】快捷键启动 Eclipse 的代码辅助菜单，在辅助菜单中单击选中 System 项，便可自动输入该项。

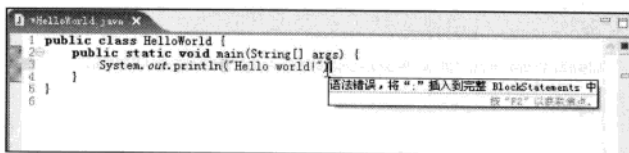


(4) 按照相同的方法，完成“System.out.println("Hello world!");”语句的输入。

(5) 如果在输入的过程中出现了漏输入或者错误的输入，将鼠标停留在红色处，编辑器还会做出正确的语法提示。



(6) 如果完成了完整语法的输入，最后没有输入【;】语句结束符号，系统也会给出正确的语法提示。



(7) 完整的代码如下。

```
01 public class HelloWorld {
02     public static void main(String[] args) {
03         System.out.println("Hello world!"); //输出文中信息到控制台
04     }
05 }
```

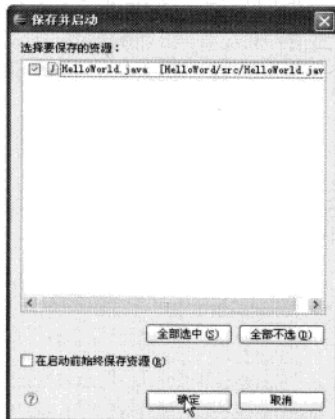
3.2.4 运行 Java 程序

【范例 3-4】 运行 Hello Word 程序。在控制台中显示 Hello Word 程序运行结果。

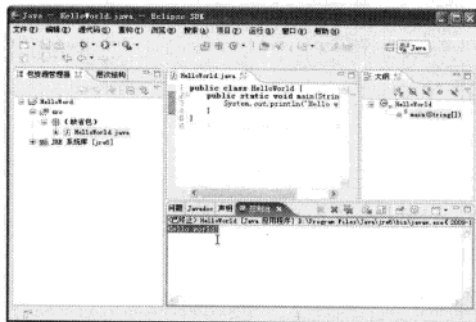
前面所创建的 HelloWorld 类是包含 main()主方法的，是一个可以运行的主类。具体运行方法如下。

❶ 在【包资源管理器】窗口中，右击【HelloWorld.java】Java 源文件。

- ② 在弹出的快捷菜单中执行【运行方式】>【Java 应用程序】命令，在弹出的【保存并启动】对话框中单击【确定】按钮，保存并启动应用程序。



- ③ 单击【确定】按钮后，程序运行结果便可在控制台中显示出来。

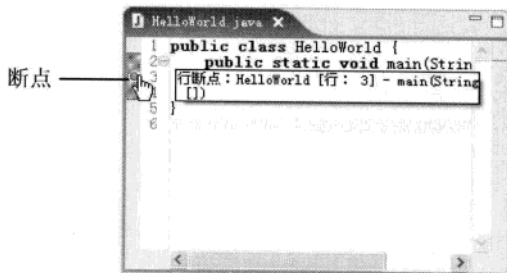


3.3 在 Eclipse 中调试程序

▶ 本节视频教学录像：21 分钟

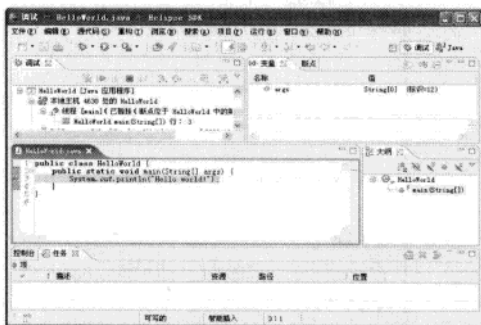
在 Eclipse 中，交互式运行代码是其最强大的特性之一。使用 JDT 调试器，可以逐行执行 Java 程序，检查程序不同位置变量的值，这个过程在定位代码中的问题时非常有用。

为了准备调试，需要在代码中设置一个断点，以便让调试器暂停执行，而允许进行调试，否则程序会从头执行到尾，就没有机会调试了。在编辑器左边灰色边缘双击，这里将第 3 行代码 `println()` 函数位置设置为断点，此时会显示一个蓝色的小点，表示一个活动的断点。

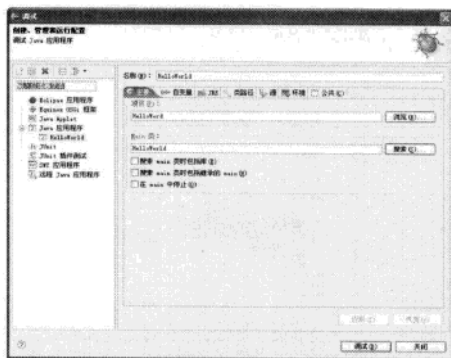


在调试器下运行程序和运行它类似，Eclipse 提供了两个选项：**【Java 应用程序】**和**【调试】**。在调试时要确保编辑器中的 HelloWorld 代码被全部选中。

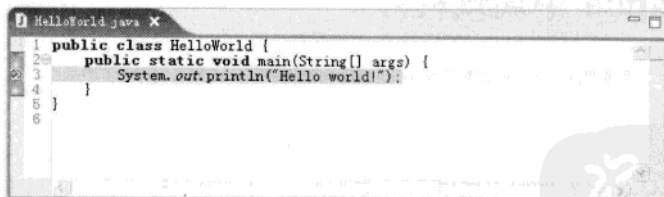
(1) 选中全部代码并右击，在弹出的快捷菜单中执行**【调试方式】>【Java 应用程序】**命令，Eclipse 将会启动程序，切换到调试透视图，在断点暂停执行。



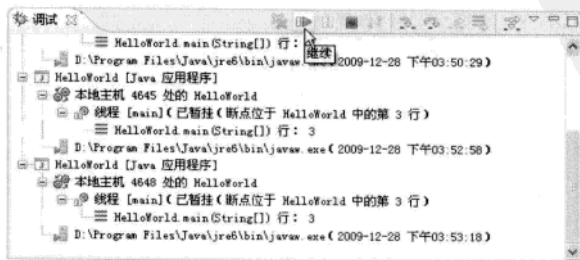
(2) 选中全部代码并右击，在弹出的快捷菜单中执行**【调试方式】>【调试】**命令，打开**【调试】**对话框。



(3) 单击**【调试】**按钮，Eclipse 将会启动程序，切换到调试透视图，在断点暂停执行。



程序执行到断点被暂停后，可以通过**【调试】**窗口工具栏中的按钮进行相应的调试操作，例如继续、停止等。



3.4 练一练

一、填空题

1. Eclipse 是_____花巨资开发的 IDE 集成开发环境 (Integrated Development Environment)。
2. 【源代码】菜单中所包含的命令都是和代码编写相关的, 主要用于_____工作。
3. 选择【文件】>【_____】>【项目...】命令, 打开【新建项目】对话框。

二、简答题

1. Eclipse 提供了哪两种程序调试方式?
2. 取消 Eclipse 启动时出现的设置工作空间的对话框的具体方法是什么?

3.5 跟我上机

在 Eclipse 中创建一个项目并编写程序, 实现在控制台输出“《Java 从入门到精通》是学习 Java 的最佳宝典”。



第4章

最常用的编程元素——常量与变量



本章视频教学录像：14 分钟

Java语言强大灵活，与C语言语法有很多相似的地方。要想熟练使用Java语言进行程序开发，就必须从了解Java语言基础开始。本章讲解Java中常量和变量的声明与应用、变量的命名规则以及作用范围等。本章内容是接下来所有章节的基础，初学者应该认真学习。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握常量和变量的声明方法
- ☐ 掌握变量的命名规则
- ☐ 掌握变量的作用范围
- ☐ 掌握常量和变量的应用技巧



4.1 常量

▶ 本节视频教学录像：14 分钟

常量就是固定不变的量，一旦被定义，它的值就不能再被改变。

4.1.1 声明常量

声明常量的语法为：

```
final 数据类型 常量名称[=值]
```

常量名称通常使用大写字母，例如 PI、YEAR 等，但并不是硬性要求，仅仅是一个习惯而已。

4.1.2 常量应用示例

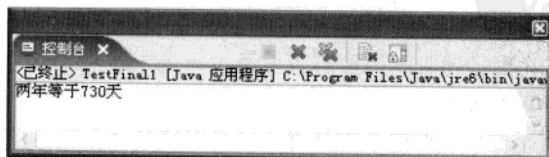
当常量用于一个类的成员变量时，必须给常量赋值，否则会出现编译错误。

【范例 4-1】 声明一个常量用于成员变量（代码 4-1.java）。

```
public class TestFinal1 {  
    static final int YEAR = 365;  
    public static void main(String[] args) {  
        System.out.println("两年等于"+2*YEAR+"天");  
    }  
}
```

【运行结果】

保存并运行程序，结果如图所示。



4.2 变量

变量是利用声明的方式，将内存中的某个块保留下来以供程序使用。可以声明为块记载的数

据类型为整型、字符型、浮点型或是其他的数据类型，以作为变量保存之用。

4.2.1 声明变量

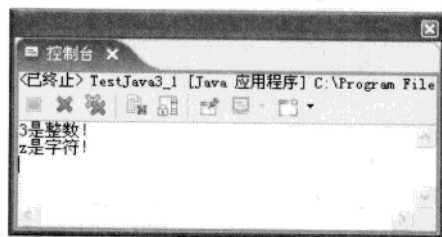
下面先来看一个简单的实例，以便了解 Java 里变量与常量之间的关系。在下面的程序里声明了两种 Java 经常使用到的变量，分别为整型变量 `num` 与字符变量 `ch`。为它们赋值后，再把它们的值分别在显示器上显示。

【范例 4-2】 声明两个变量，一个是整型，另一个是字符型（代码 4-2.java）。

```
01      // 下面的程序声明了两个变量，一个是整型，另一个是字符型
02      public class TestJavaintchar
03      {
04          public static void main(String args[])
05          {
06              int num = 3;                // 声明一整型变量 num，赋值为 3
07              char ch = 'z';            // 声明一字符变量 ch，赋值为 z
08              System.out.println(num+ "是整数! ");    // 输出 num 的值
09              System.out.println(ch + "是字符! ");    // 输出 ch 的值
10          }
11      }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

在 `TestJava 3_1` 中，声明了两种不同类型的变量 `num` 与 `ch`，并分别将常量 3 与字符“z”赋值给这两个变量，最后将它们显示在显示器上。

声明一个变量时，编译程序会在内存里开辟一块足以容纳此变量的内存空间给它。不管变量的值如何改变，都永远使用相同的内存空间。因此，善用变量是一种节省内存的方式。

常量是不同于变量的一种类型，它的值是固定的，例如整数常量、字符串常量。通常给变量赋值时，会将常量赋值给它。在程序 `TestJava 3_1` 中，第 6 行 `num` 是整型变量，而 3 则是常量。此行的作用是声明 `num` 为整型变量，并把常量 3 这个值赋给它。与此相同，第 7 行声明了一个字符变量 `ch`，并将字符常量“z”赋给它。当然，在程序编写的过程中，可以为变量重新赋值，也可以使用已经声明过的变量。

4.2.2 变量的命名规则

变量也是一种标识符，所以它也遵循标识符的命名规则。

- (1) 变量名可由任意顺序的大小写字母、数字、下划线 (_) 和美元符号 (\$) 等组成。
- (2) 变量名不能以数字开头。
- (3) 变量名不能是 Java 中的保留关键字。

4.2.3 变量的作用范围

变量是有作用范围的，一旦超出它的范围，就无法再使用这个变量。例如张三在 A 村很知名。你打听 A 村的张三，人人都知道，可你到 B 店打听，就没人知道。也就是说，在 B 店张三是无法访问的。就算碰巧 B 店也有个叫张三的，但此张三已经非彼张三了。

按作用范围进行划分，变量分为成员变量和局部变量。

1. 成员变量

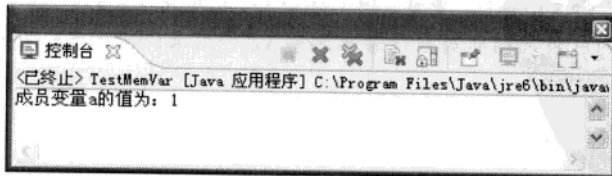
在类体中定义的变量为成员变量。它的作用范围为整个类，也就是说在这个类中都可以访问到定义的这个成员变量。

【范例 4-3】 探讨成员变量的作用范围（代码 4-3.java）。

```
public class TestMemVar {  
  
    static int a = 1;           //定义一个成员变量  
    public static void main(String[] args) {  
        System.out.println("成员变量 a 的值为: "+ a);  
    }  
}
```

【运行结果】

保存并运行程序，结果如图所示。



2. 局部变量

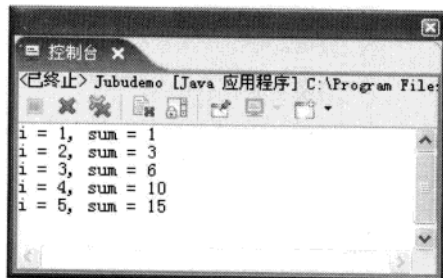
Java 可以在程序的任何地方声明变量，当然也可以在循环里声明。有趣的是，在循环里声明的变量只是局部变量，只要跳出循环，这个变量便不能再使用。下面用一个范例来说明局部变量的使用方法。

【范例 4-4】 局部变量的使用（代码 4-4.java）。

```
01 // 以下程序说明了局部变量的使用方法
02 public class Jubudemo
03 {
04     public static void main(String[] args)
05     {
06         int sum = 0 ;
07         // 下面是 for 循环的使用，计算 1~5 数字累加之和
08         for(int i=1;i<=5;i++)
09         {
10             sum = sum + i ;
11             System.out.println("i = "+i+", sum = "+sum);
12         }
13     }
14 }
```

【运行结果】

保存并运行程序，结果如图所示。

**【代码详解】**

把变量 *i* 声明在 for 循环里，因此变量 *i* 在此就是局部变量，它的有效范围仅在 for 循环内（8~12 行），只要一离开这个循环，变量 *i* 便无法使用。相对而言，变量 *sum* 是声明在 *main()* 方法的开始处，因此它的有效范围从第 6 行开始到第 12 行结束，当然，for 循环内也属于变量 *sum* 的有效范围。

4.3 练一练

一、填空题

- _____ 是利用声明的方式，将内存中的某个块保留下来以供程序使用。
- _____ 就是固定不变的量，一旦被定义，它的值就不能再被改变。
- 当常量用于一个类的成员变量时，必须给常量 _____，否则会出现编译错误。

二、简答题

1. 简述变量的命名规则。
2. 按作用范围划分，变量是如何划分的？

4.4 跟我上机

编写一个定义局部变量的简单程序。



第 5 章

不可不知的数据分类法——数据类型



本章视频教学录像：21 分钟

Java的数据类型在程序语言的构成要素里占有相当重要的地位。Java的数据类型可分为基本数据类型与引用数据类型两大类。本章介绍Java的基本数据类型和数据类型的转换，其中重点讲述整数、浮点类型、字符类型和布尔类型。最后对基本数据类型的默认值进行专题研究。

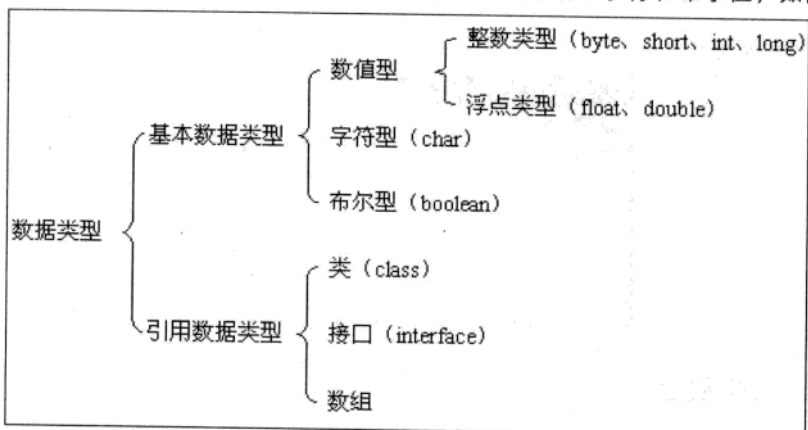
本章要点（已掌握的在方框中打勾）

- ☐ 掌握整数类型的分类
- ☐ 掌握浮点数据类型的分类
- ☐ 掌握字符类型数据
- ☐ 掌握布尔类型数据
- ☐ 熟悉基本数据类型默认值的意义与使用方法



5.1 整数类型

在 Java 中规定了 8 种基本数据类型变量来存储整数、浮点、字符和布尔值，如图所示。



Java 的基本数据类型如表所示。

数据类型	字节	可表示的数据范围
long (长整数)	64	-9223372036854775808 ~ 9223372036854775807
int (整数)	32	-2147483648 ~ 2147483647
short (短整数)	16	-32768 ~ 32767
byte (位)	8	-128 ~ 127
char (字符)	2	0 ~ 255
float (单精度)	32	-3.4E38 (-3.4×10^{38}) ~ 3.4E38 (3.4×10^{38})
double (双精度)	64	-1.7E308 (-1.7×10^{308}) ~ 1.7E308 (1.7×10^{308})

举例来说，想声明一个短整型变量 sum 时，可以在程序中做出如下的声明。

```
short sum; // 声明 sum 为短整型
```

经过声明之后，Java 即会在可使用的内存空间中，寻找一个占有 2 个字节的块供 sum 变量使用，同时这个变量的范围只能在 -32768 ~ 32767 之间。

5.1.1 byte 类型

在 Java 中，byte 类型占据 1 个字节内存空间，数据的取值范围为 -128 ~ 127。

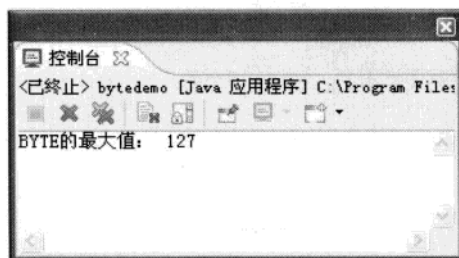
【范例 5-1】 byte 类型数据的使用（代码 5-1.java）。

```

01 public class bytedemo
02 {
03     public static void main(String args[])
04     {
  
```

```
05     byte byte_max = Byte.MAX_VALUE;    // 得到 Byte 型的最大值
06
07     System.out.println("BYTE 的最大值: "+byte_max);
08 }
09 }
```

程序运行结果如图所示。



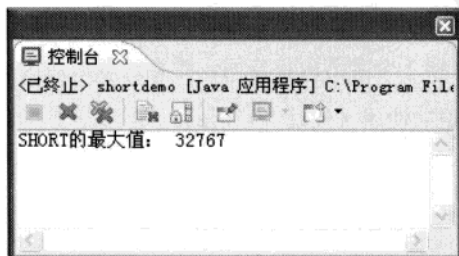
5.1.2 short 类型

short 类型数据占据 2 个字节内存空间，取值范围为 -32768 ~ 32767。

【范例 5-2】 short 类型数据的使用（代码 5-2.java）。

```
01     public class shortdemo
02     {
03         public static void main(String args[])
04         {
05             short short_max = Short.MAX_VALUE;    // 得到短整型的最大值
06
07             System.out.println("SHORT 的最大值: "+short_max);
08         }
09     }
10 }
```

程序运行结果如图所示。



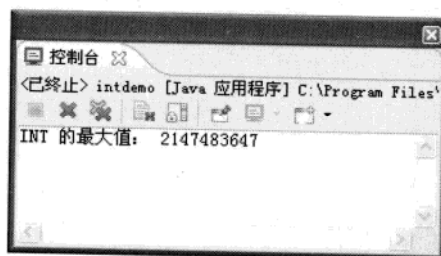
5.1.3 int 类型

int 类型数据占据 4 个字节内存空间，取值范围为-2147483648 ~ 2147483647。

【范例 5-3】 int 类型数据的使用（代码 5-3.java）。

```
01 public class intdemo
02 {
03     public static void main(String args[])
04     {
05         int int_max = java.lang.Integer.MAX_VALUE ;        // 得到整型的最大值
06
07         System.out.println("INT 的最大值: "+int_max);
08
09     }
10 }
```

程序运行结果如图所示。



5.1.4 long 类型

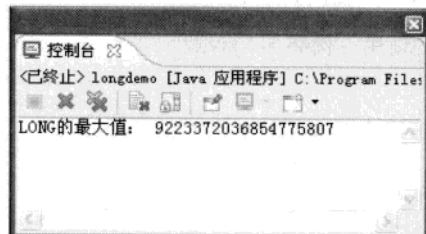
long 类型数据占据 8 个字节内存空间，取值范围为 -9223372036854775808 ~ 9223372036854775807。

【范例 5-4】 long 类型数据的使用（代码 5-4.java）。

```
01 public class longdemo
02 {
03     public static void main(String args[])
04     {
05         long long_max = java.lang.Long.MAX_VALUE ;        // 得到长整型的最大值
06
07         System.out.println("LONG 的最大值: "+long_max);
08
09     }
```


10 }

程序运行结果如图所示。



5.2 浮点类型

Java 浮点数据类型主要有双精度 double 和单精度 float 两个类型。

double 类型：共 8 个字节，64 位，第 1 位为符号位，中间 11 位表示指数，最后 52 位表示尾数。

float 类型：共 4 个字节，32 位，第 1 位为符号位，中间 8 位表示指数，最后 23 位表示尾数。

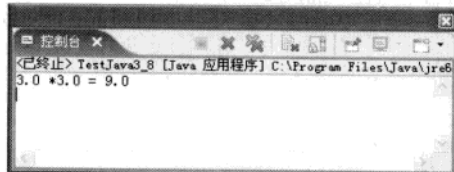
5.2.1 float 类型

下面举一个简单的例子。在程序中，声明一个 float 类型的变量 num，并赋值为 3.0，将 num*num 的运算结果输出到显示器上。

【范例 5-5】 浮点类型的使用（代码 5-5.java）。

```
01 // 下面这段程序说明了浮点数类型的使用方法
02 public class TestJava3_8
03 {
04     public static void main(String args[])
05     {
06         float num = 3.0f;
07         System.out.println(num+"**"+num+"="+num*num);
08     }
09 }
```

程序运行结果如图所示。



5.2.2 double 类型

当浮点数的表示范围不够大的时候,还有一种双精度(double)浮点数可供使用。双精度浮点数类型的长度为 64 个字节,有效范围为 -1.7×10^{308} 到 1.7×10^{308} 。

Java 提供了浮点数类型的最大值与最小值的代码,其所使用的类全名与所代表的值的范围,可以在下表中查阅。

类别	float	double
使用类全名	java.lang.Float	java.lang.Double
最大值	MAX_VALUE	MAX_VALUE
最大值常量	3.4028235E38	107976931348623157E308
最小值	MIN_VALUE	MIN_VALUE
最小值常量	1.4E-45	4.9E-324

相同的,在类全名中,可以省去类库 java.lang,直接取用类名称即可。下面的程序是输出 float 与 double 两种浮点数类型的最大值与最小值,读者可以将下面程序的输出结果与上表一一进行比较。

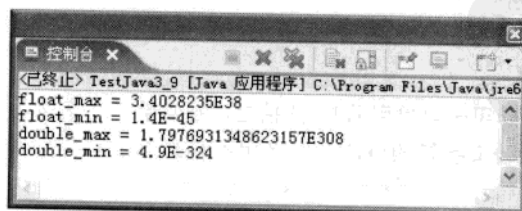
【范例 5-6】 取得单精度和双精度浮点数类型的最大、最小值(代码 5-6.java)。

```

01 // 下面这段程序用于取得单精度和双精度浮点数类型的最大、最小值
02 public class TestJava3_9
03 {
04     public static void main(String args[])
05     {
06         System.out.println("float_max = "+java.lang.Float.MAX_VALUE);
07         System.out.println("float_min = "+java.lang.Float.MIN_VALUE);
08         System.out.println("double_max = "+java.lang.Double.MAX_VALUE);
09         System.out.println("double_min = "+java.lang.Double.MIN_VALUE);
10     }
11 }

```

程序运行结果如图所示。



下列为声明与设置 float 与 double 类型的变量时应注意的事项。

```

double num1 = -6.3e64 ;           // 声明 num1 为 double, 其值为 -6.3 1064
double num2 = -5.34E16 ;         // e 也可以用大写的 E 来取代
float num3 = 7.32f ;             // 声明 num3 为 float, 并设初值为 7.32f

```

```
float num4 = 2.456E67; // 错误, 因为 2.456 1067 已超过 float 可表示的范围
```

5.3 字符类型

▶ 本节视频教学录像: 13 分钟

字符类型在内存中占有 2 个字节, 定义时语法为:

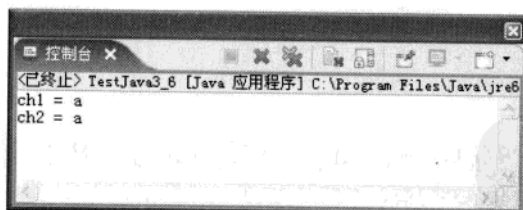
```
char a = '字符'
```

下面给出一个范例来说明它的用法。

【范例 5-7】 直接给字符类型赋值 (代码 5-7.java)。

```
01 // 字符类型也可以直接赋给数值, 下面的这段程序就是采用这种赋值方式
02 public class TestJava3_6
03 {
04     public static void main(String args[])
05     {
06         char ch1 = 97;
07         char ch2 = 'a';
08
09         System.out.println("ch1 = "+ch1);
10         System.out.println("ch2 = "+ch2);
11     }
12 }
```

程序运行结果如图所示。



需要注意的是: 字符要用一对单引号 (') 括起。但如果把一个字符变量赋值成一个单引号, 就会出问题, 这样就有了转义字符的概念。应当用不容易混淆的字符来代替那些敏感字符。

下表为常用的转义字符。

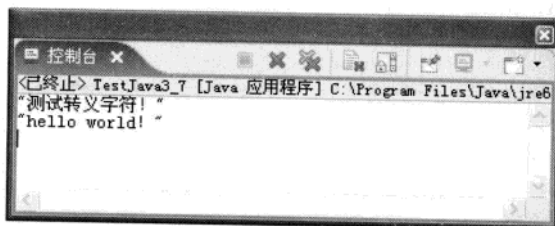
转义字符	所代表的意义	转义字符	所代表的意义
\f	换页	\\	反斜线
\b	倒退一格	\'	单引号
\r	归位	\"	双引号
\t	跳格	\n	换行

以下的程序为例，将 `ch` 赋值为 `"\"`（要以单引号 `'` 包围），并将字符变量 `ch` 输出在显示器上，同时在打印的字符串里直接加入转义字符，读者可自行比较两种方式的差异。

【范例 5-8】 转义字符的使用（代码 5-8.txt）。

```
01 // 下面这段程序表明了转义字符的使用方法
02 public class TestJava3_7
03 {
04     public static void main(String args[])
05     {
06         char ch = "\"";
07
08         System.out.println(ch+"测试转义字符! "+ch);
09         System.out.println("\"hello world! \"");
10     }
11 }
```

程序运行结果如图所示。



不管是用变量存放转义字符，或是直接使用转义字符的方式来输出字符串，程序都可以顺利运行。由于使用变量会占用内存资源，因此似乎显得有些不妥；当然也可以不必声明字符变量就输出转义字符，但如果在程序中加上太多的转移字符，以至于造成混淆而不易阅读时，利用声明字符变量的方式就是一个很好的选择。

5.4 布尔类型

本节视频教学录像：4 分钟

布尔（boolean）类型的变量，只有 `true`（真）和 `false`（假）两种。也就是说，当将一个变量定义成布尔类型时，它的值只能是 `true` 或 `false`，除此之外，没有其他的值可以赋值给这个变量。举例来说，想声明名称为 `status` 的变量为布尔类型，并设置为 `true` 值，可以使用下面的语句。

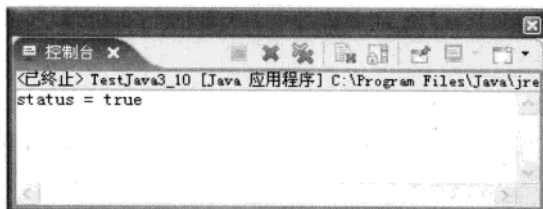
```
boolean status = true; // 声明布尔变量 status，并赋值为 true
```

经过声明之后，布尔变量的初值即为 `true`，当然如果在程序中需要更改 `status` 的值时，即可随时更改。将上述的内容写成了程序 `TestJava3_10`，读者可以先熟悉一下布尔变量的使用。

【范例 5-9】 布尔值类型变量的声明（代码 5-9.java）。

```
01    // 下面的程序声明了一个布尔值类型的变量
02    public class TestJava3_10
03    {
04        public static void main(String args[])
05        {
06            // 声明一布尔型的变量 status，布尔型只有两个取值：true、false
07            boolean status = true ;
08            System.out.println("status = "+status);
09        }
10    }
```

程序运行结果如图所示。



布尔值通常用来控制程序的流程，读者可能会觉得有些抽象，本书会陆续在后面的章节中介绍布尔值在程序流程中所起的作用。

5.5 数据类型的转换

Java 有严格的数据类型限制。数据类型是不可以轻易转换的。但在特殊情况下还是需要进行这样的操作，但必须有严格的步骤和规定。数据类型的转换方式可分为“自动类型转换”及“强制类型转换”两种。

5.5.1 自动类型转换

在程序中已经定义好了数据类型的变量，若想用另一种数据类型表示时，Java 会在下列的条件皆成立时，自动进行数据类型的转换。

- (1) 转换前的数据类型与转换后的类型兼容。
- (2) 转换后的数据类型的表示范围比转换前的类型大。

以“扩大转换”来看可能比较容易理解——字符与整数是可使用自动类型转换的；整数与浮点数亦是兼容的；但是由于 boolean 类型只能存放 true 或 false，与整数及字符不兼容，因此不能进行类型的转换。接下来看看当两个数中有一个为浮点数时，其运算的结果会有什么样的变化。

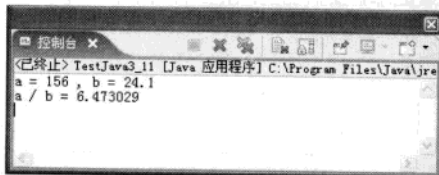
【范例 5-10】 声明两个变量，一个是整型，另一个是浮点型（代码 5-10.java）。

```

01 // 下面这段程序声明了两个变量，一个是整型，另一个是浮点型
02 public class TestJava3_11
03 {
04     public static void main(String args[])
05     {
06         int a = 156 ;
07         float b = 24.1f ;           // 声明一浮点型变量 f，并赋值
08
09         System.out.println("a = "+a+" , b = "+b);
10         System.out.println("a / b = "+(a/b));      // 在这里整型会自动转换为浮点型
11     }
12 }

```

程序运行结果如图所示。



从运行的结果可以看出，当两个数中有一个为浮点数时，其运算的结果会直接转换为浮点数。当表达式中变量的类型不同时，Java 会自动以较小的表示范围转换成较大的表示范围后，再作运算。也就是说，假设有一个整数和双精度浮点数作运算，Java 会把整数转换成双精度浮点数后再作运算，运算结果也会变成双精度浮点数。关于表达式的数据类型转换，在后面的章节中会有更详细的介绍。

5.5.2 强制类型转换

当程序需要转换数据类型时，可实施强制性的类型转换，其语法如下。

（欲转换的数据类型）变量名称；

下面的程序说明了在 Java 里，整数与浮点数是如何转换的。

【范例 5-11】 自动转换和强制转换的使用方法（代码 5-11.java）。

```

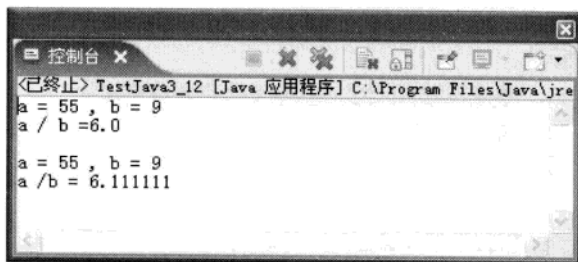
01 // 下面的范例说明了自动转换和强制转换这两种转换的使用方法
02 public class TestJava3_12
03 {
04     public static void main(String args[])
05     {

```

```

06      int a = 55 ;
07      int b = 9 ;
08      float g,h ;
09
10      System.out.println("a = "+a+" , b = "+b);
11      g = a/b ;
12      System.out.println("a / b ="+g+"\n");
13      System.out.println("a = "+a+" , b = "+b);
14      h = (float)a/b ;           //在这里对数据类型进行强制转换
15      System.out.println("a /b = "+h);
16  }
17  }
    
```

程序运行结果如图所示。



当两个整数相除时，小数点以后的数字会被截断，使得运算的结果保持为整数。但由于这并不是预期的计算结果，而想要得到运算的结果为浮点数，就必须将两个整数中的一个（或是两个）强制转换为浮点数，下面的 3 种写法都正确。

<code>(float)a/b;</code>	// 将整数 <code>a</code> 强制转换成浮点数，再与整数 <code>b</code> 相除
<code>a/(float)b;</code>	// 将整数 <code>b</code> 强制转换成浮点数，再以整数 <code>a</code> 除之
<code>(float)a/(float)b;</code>	// 将整数 <code>a</code> 与 <code>b</code> 同时强制转换成浮点数，再相除

只要在变量前面加上欲转换的数据类型，运行时就会自动将此行语句里的变量做类型转换的处理，但这并不影响原先所定义的数据类型。

此外，若是将一个超出该变量可表示范围的值赋给这个变量，这种转换称为缩小转换。由于在转换的过程中可能会丢失数据的精确度，因此 Java 并不会自动做这些类型的转换，此时就必须要做强制性的转换。

5.6 专题研究——基本数据类型的默认值

▶ 本节视频教学录像：4 分钟

在 Java 中，若在变量的声明时没有赋初值，则会给该变量赋默认值。下表列出了各种类型的默认值。

数据类型	默认值
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0d
char	\u0000 (空)
boolean	false

在某些情形下,Java 会给予这些没有赋初始值的变量一个确切的默认值,但这没有任何意义,也没有必要,但应该保证程序执行时,不会有这种未定义值的变量存在。虽然这种方式给程序编写者带来了很大便利,但是过于依赖系统给变量赋初值,就不容易检测到是否已经给予变量应有的值了,这是个需要注意的问题。

5.7 练一练

一、填空题

1. Java 数据类型分为_____数据类型和_____数据类型两大类。
2. int 类型数据占据_____个字节内存空间,取值范围为_____。

二、简答题

简述在 Java 中,数据类型转换的规则。

5.8 跟我上机

编写程序,运行后输出 long 类型数据的最小数值。

邮
电

第6章

最重要的编程部件——运算符、表达式与语句



本章视频教学录像：1 小时 18 分钟

运算符、表达式和语句是编程的主要部件，能够使系统程序直接对内存进行操作，从而大大提高程序的执行能力。本章介绍Java运算符的用法、表达式与运算符之间的关系，以及表达式里各种变量的数据类型的转换等。学完本章，希望读者能对Java语句的运作过程有更深一层的认识。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握各种运算符的用法
- ☐ 掌握各种表达式的用法
- ☐ 掌握表达式与运算符的关系
- ☐ 掌握表达式中数据类型的转换技巧

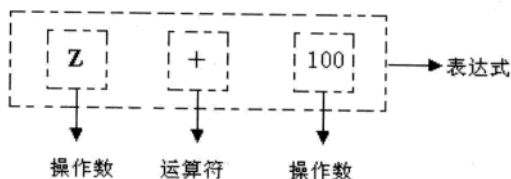


6.1 运算符

▶ 本节视频教学录像：42 分钟

程序是由许多语句组成的，而语句的基本单位就是表达式与运算符。Java 的运算符可分为 4 类：算术运算符、关系运算符、逻辑运算符和位运算符。

Java 中的语句有多种形式，表达式就是其中的一种形式。表达式是由操作数与运算符所组成：操作数可以是常量、变量，也可以是方法，而运算符就是数学中的运算符号，如“+”、“-”、“*”、“/”、“%”等。例如下面的表达式（ $z+100$ ），“ z ”与“100”都是操作数，而“+”就是运算符。



6.1.1 赋值运算符

下面通过一个范例来讲解赋值运算符的用法。

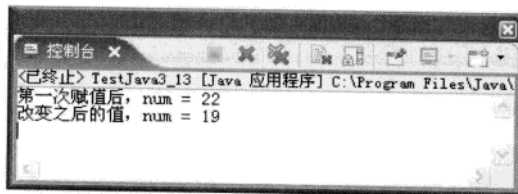
【范例 6-1】 在程序中用“=”赋值（代码 6-1.java）。

```

01 // 在程序中赋值采用“=”
02 public class TestJava3_13
03 {
04     public static void main(String args[])
05     {
06         int num = 22;           // 声明整数变量 num，并赋值为 22
07
08         System.out.println("第一次赋值后，num = "+num); // 输出 num 的值
09
10         num = num - 3;          // 将变量 num 的值减 3 之后再赋给 num 变量
11         System.out.println("改变之后的值，num = "+num); // 输出后 num 的值
12     }
13 }

```

程序运行结果如图所示。



当然，在程序中也可以将等号后面的值赋给其他的变量，例如：

```
in sum = num1+num2;           // num1 与 num2 相加之后的值再赋给变量 sum 存放
```

num1 与 num2 的值经过运算后仍然保持不变，sum 会因为“赋值”的操作而更改内容。

6.1.2 一元运算符

下表列出了一元运算符的成员。

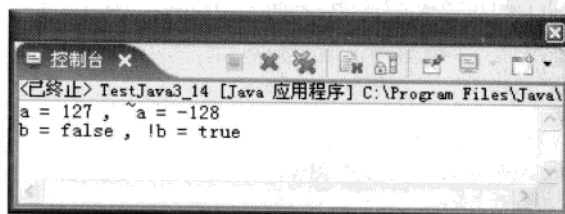
一元运算符	意义
+	正号
-	负号
!	NOT, 否
~	取补码

下面的程序声明了 byte 类型的变量 a 及 boolean 类型的变量 b，可以看到这两个变量分别进行了“~”与“!”运算。

【范例 6-2】 一元运算符的使用（代码 6-2.java）。

```
01 // 下面这段程序说明了一元运算符的使用
02 public class TestJava3_14
03 {
04     public static void main(String args[])
05     {
06         byte a = java.lang.Byte.MAX_VALUE; // 声明并将其类型最大值赋给 a
07         boolean b = false;
08
09         System.out.println("a = "+a+", ~a = "+(~a));
10         System.out.println("b = "+b+", !b = "+(!b));
11     }
12 }
```

程序运行结果如图所示。



【代码详解】

第 6 行声明了 byte 变量 a，并赋值为该类型的最大值，即 a 的值为 127。程序第 7 行声明了 boolean 变量 b，赋值为 false。

第 9 行输出 a 与 ~a 的运算结果。

第 10 行输出 b 与 !b 的运算结果。b 的值为 false，因此进行 “!” 运算后，b 的值就变成了 true。

6.1.3 算术运算符

算术运算符在数学上经常会用到，下表列出了它的成员。

算术运算符	意义
+	加法
-	减法
*	乘法
/	除法
%	余数

(1) 加法运算符 “+”

将加法运算符 “+” 的前后两个操作数相加，如下面的语句。

```
System.out.println("3 + 8 = "+(3+8));    // 直接输出表达式的值
```

(2) 减法运算符 “-”

将减法运算符 “-” 前面的操作数减去后面的操作数，如下面的语句。

```
num = num - 3;           // 将 num-3 运算之后赋值给 num 存放
a = b - c;               // 将 b - c 运算之后赋值给 a 存放
120 - 10;               // 运算 120 - 10 的值
```

(3) 乘法运算符 “*”

将乘法运算符 “*” 的前后两个操作数相乘，如下面的语句。

```
b = b * 5;              // 将 b*5 运算之后赋值给 b 存放
a = a * a;              // 将 a * a 运算之后赋值给 a 存放
19 * 2;                 // 运算 19 * 2 的值
```

(4) 除法运算符 “/”

将除法运算符 “/” 前面的操作数除以后面的操作数，如下面的语句。

```
a = b / 5;              // 将 b / 5 运算之后的值赋值给 a 存放
c = c / d;              // 将 c / d 运算之后的值赋值给 c 存放
15 / 5;                 // 运算 15 / 5 的值
```

使用除法运算符时要特别注意一点，就是数据类型的问题。上面的例子来说，当 a、b、c、d 的类型皆为整数，若是运算的结果不能整除时，输出的结果与实际的值就会有差异，这是因为整数类型的变量无法保存小数点后面的数据所致，因此在声明数据类型及输出时要特别小心。以下的程序为例，在程序里给两个整型变量 a、b 赋值，并将 a / b 的运算结果输出。

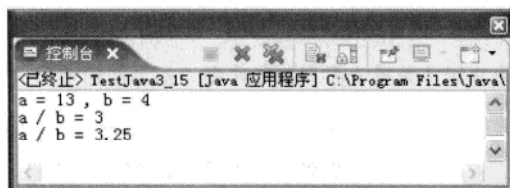
【范例 6-3】 除法运算符的使用（代码 6-3.java）。

```

01 // 下面这段程序说明了除法运算符的使用方法
02 public class TestJava3_15
03 {
04     public static void main(String[] args)
05     {
06         int a = 13 ;
07         int b = 4 ;
08
09         System.out.println("a = "+a+" , b = "+b);
10         System.out.println("a / b = "+(a/b));
11         System.out.println("a / b = "+((float)a/b));    // 进行强制类型转换
12     }
13 }

```

程序运行结果如图所示。



【代码详解】

第 10 行与 11 行，程序分别做出了不同的输出。在第 10 行，因为 a、b 皆为整数类型，输出结果也会是整数类型，程序运行结果与实际的值不同。

在第 11 行，为了保证程序运行结果与实际的值相同，使用了强制性的类型转换，即将整数类型（int）转换成浮点数类型（float），程序运行的结果才不会有问题。

(5) 余数运算符“%”

将余数运算符“%”前面的操作数除以后面的操作数，取其所得到的余数。下面的语句是余数运算符的使用范例。

```

num = num % 3 ;           // 将 num%3 运算之后赋值给 num 存放
a = b % c ;               // 将 b%c 运算之后赋值给 a 存放
100 % 7 ;                 // 运算 100%7 的值

```

以下的程序为例，声明两个整型变量 a、b，并分别赋值为 5 和 3，再输出 a%b 的运算结果。

【范例 6-4】 取模操作（代码 6-4.java）。

```

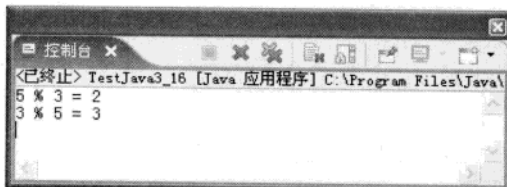
01 // 在 JAVA 中用%进行取模操作
02 public class TestJava3_16

```

```

03  {
04      public static void main(String[] args)
05      {
06          int a = 5 ;
07          int b = 3 ;
08
09          System.out.println(a+" % "+b+" = "+(a%b));
10          System.out.println(b+" % "+a+" = "+(b%a));
11      }
12  }
    
```

程序运行结果如图所示。



6.1.4 关系运算符与 if 语句

if 语句的格式如下。

```

if (判断条件)
    语句 ;
    
```

如果括号中的判断条件成立，就会执行后面的语句；若是判断条件不成立，则后面的语句就不会被执行。如下面的程序片段。

```

if (x>0)
    System.out.println("I like Java ! ");
    
```

当 x 的值大于 0，就是判断条件成立时，会执行输出字符串“I like Java!”的操作；相反，当 x 的值为 0 或是小于 0 时，if 语句的判断条件不成立，就不会进行上述操作。下表列出了关系运算符的成员，这些运算符在数学上也是经常使用的。

关系运算符	意义
>	大于
<	小于
>=	大于等于
<=	小于等于
==	等于
!=	不等于

在 Java 中，关系运算符的表示方式和在数学中类似，但是由于赋值运算符为“=”，为了避免混淆，当使用关系运算符“等于”(==)时，就必须用 2 个等号表示；而关系运算符“不等于”

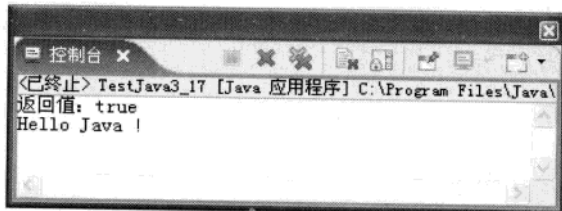
的形式有些特别，用“!=”代表，这是因为在键盘上想要取得数学上的不等于符号“≠”较为困难，所以就用“!=”表示不等于。

当使用关系运算符去判断一个表达式的成立与否时，若是判断式成立则会产生一个响应值 true，若是判断式不成立，则会产生响应值 false。以下面的程序为例，判断 if 语句括号中的条件是否成立，若是成立则执行 if 后面的语句。

【范例 6-5】 关系运算符的使用（代码 6-5.java）。

```
01 // 下面这段程序说明了关系运算符的使用方法，关系运算符返回值为布尔值
02 public class TestJava3_17
03 {
04     public static void main(String[] args)
05     {
06         if(5>2)
07             System.out.println("返回值: "+(5>2));
08
09         if(true)
10             System.out.println("Hello Java !");
11
12         if((3+6)==(3-6))
13             System.out.println("I like Java !");
14     }
15 }
```

程序运行结果如图所示。



【代码详解】

在第 6 行中，由于 $5>2$ 的条件成立，所以执行第 7 行的语句：输出返回值 true。

在第 9 行中，若是 if 语句的参数为 true，判断亦成立，所以接着执行第 10 行的语句：输出字符串“Hello TestJava!”。

在第 12 行中， $3+6$ 并不等于 $3-6$ ，if 的判断条件不成立，所以第 13 行语句不被执行。

6.1.5 递增与递减运算符

递增与递减运算符在 C/C++ 中就已经存在了，Java 仍然将它们保留了下来，这是因为它们具有相当大的便利性。下表列出了递增与递减运算符的成员。

递增与递减运算符	意义
++	递增，变量值加 1
--	递减，变量值减 1

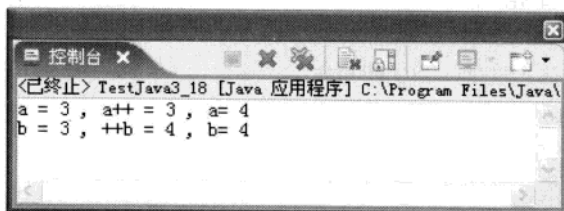
【范例 6-6】 “++” 运算符的两种使用方法（代码 6-6.java）。

```

01 // 下面这段程序说明了 “++” 的两种用法
02 public class TestJava3_18
03 {
04     public static void main(String args[])
05     {
06         int a = 3, b = 3;
07
08         System.out.print("a = "+a);           // 输出 a
09         System.out.println(" , a++ = "+(a++)+", a= "+a); // 输出 a++和 a
10         System.out.print("b = "+b);           // 输出 b
11         System.out.println(" , ++b = "+(++b)+", b= "+b); // 输出 ++b 和 b
12     }
13 }

```

程序运行结果如图所示。



【代码详解】

在第 9 行中，输出 a++及运算后的 a 的值，所以执行完 a++后，a 的值才会加 1，变成 4。

在第 11 行中，输出 ++b 运算后 b 的值，所以执行 ++b 前，b 的值即先加 1，变成 4。

同样，递减运算符 “--” 的使用方式和递增运算符 “++” 是相同的，递增运算符 “++” 用来将变量值加 1，而递减运算符 “--” 则是用来将变量值减 1。此外，递增与递减运算符只能将变量加 1 或减 1，若是想要将变量加减非 1 的数时，还是得用原来的 “a = a+2” 的方法。

6.1.6 逻辑运算符

在 if 语句中也可以看到逻辑运算符，下表列出了它的成员。

逻辑运算符	意义
&&	AND，与
	OR，或

下面是使用逻辑运算符的例子。

`a>0 && b>0` // 两个操作数皆为真，运算结果才为真
`a>0 || b>0` // 两个操作数只要一个为真，运算结果就为真

在第 1 个例子中，`a>0` 而且 `b>0` 时，表达式的返回值为 `true`，即表示这两个条件必须同时成立才行；在第 2 个例子中，只要 `a>0` 或者 `b>0`，表达式的返回值即为 `true`，这两个条件仅需要一个成立即可，读者可以参考下表中所列出的结果。

条件1	条件2	结果	
		<code>&&</code> (与)	<code> </code> (或)
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

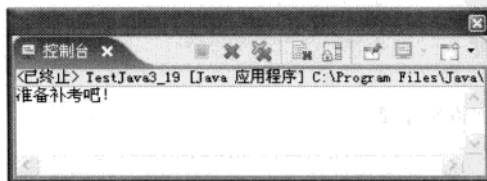
从上表中可以看出，在条件中，只要有一个条件为假 (`false`)，相与的结果就是假 (`false`)；相反，如果有一个条件为真 (`true`)，那么相或的结果就为真 (`true`)。

下面的这个程序是判断 `a` 的值是否在 0~100 之间，如果不在即表示成绩输入错误；若是 `a` 的值在 50~60 之间，则需要补考。

【范例 6-7】 在 Java 中进行与操作 (代码 6-7.java)。

```
01 // 下面这段程序是对操作进行的说明，返回值为布尔类型
02 public class TestJava3_19
03 {
04     public static void main(String[] args)
05     {
06         int a = 56 ;
07
08         if((a<0)|| (a>100))
09             System.out.println("输入的数据有错误！");
10         if((a<60)&&(a>49))
11             System.out.println("准备补考吧！");
12     }
13 }
```

程序运行结果如下。



【代码详解】

当程序执行到第 8 行时，`if` 会根据括号中 `a` 的值作判断，`a<0` 或是 `a>100` 时，条件判断成立，即会执行第 9 行的语句，输出字符串“输入的数据有错误！”。由于学生成绩是介于 0~100

分之间，因此当 a 的值不在这个范围时，就会视为是输入错误。

不管第 9 行是否执行，都会接着执行第 10 行的程序。if 再根据括号中 a 的值做判断， $a < 60$ 且 $a > 49$ 时，条件判断成立，表示该成绩需要进行补考，即会执行第 11 行的语句，输出“准备补考吧!”。

6.1.7 括号运算符

除了前面所讲的内容外，括号 () 也是 Java 的运算符，如下表所示。

括号运算符	意义
()	提高括号中表达式的优先级

括号运算符 () 是用来处理表达式的优先级的。下面是一个简单的加减乘除式子。

```
3+5+4*6-7 // 未加括号的表达式
```

相信根据读者现在所学过的数学知识，这道题应该很容易解开。按加减乘除的优先级 (*、/ 的优先级大于 +、-) 来计算，这个式子的答案为 25。但是如果先计算 $3+5+4$ 及 $6-7$ 之后再再将两数相乘，就必须将 $3+5+4$ 及 $6-7$ 分别加上括号，而成为下面的式子。

```
(3+5+4)*(6-7) // 加上括号的表达式
```

经过括号运算符 () 的运作后，计算结果为 -12，所以括号运算符 () 可以使括号内表达式的处理顺序优先。

6.1.8 运算符的优先级

下表列出了各个运算符的优先级的排列，数字越小的表示优先级越高。

优先级	运算符	类	结合性
1	()	括号运算符	由左至右
1	[]	方括号运算符	由左至右
2	!, + (正号)、- (负号)	一元运算符	由右至左
2	~	位逻辑运算符	由右至左
2	++, --	递增与递减运算符	由右至左
3	*, /, %	算术运算符	由左至右
4	+, -	算术运算符	由左至右
5	<<, >>	位左移、右移运算符	由左至右
6	>, >=, <, <=	关系运算符	由左至右
7	==, !=	关系运算符	由左至右
8	& (位运算符 AND)	位逻辑运算符	由左至右
9	^ (位运算符 XOR)	位逻辑运算符	由左至右
10	(位运算符 OR)	位逻辑运算符	由左至右
11	&&	逻辑运算符	由左至右
12		逻辑运算符	由左至右
13	?:	条件运算符	由右至左
14	=	赋值运算符	由右至左

上表的最后一栏是运算符的结合性。什么是结合性？结合性可以让程序设计者了解到运算符与操作数之间的关系及其相对位置。举例来说，当使用同一优先级的运算符时，结合性就非常重要了，它决定谁会先被处理。读者可以看看下面的例子。

```
a = b + d / 5 * 4;
```

这个表达式中含有不同优先级的运算符，其中“/”与“*”的优先级高于“+”，而“+”又高于“=”。但是读者会发现，“/”与“*”的优先级是相同的，到底d该先除以5再乘以4呢？还是5乘以4后d再除以这个结果呢？结合性的定义就解决了这方面的困扰。算术运算符的结合性为“由左至右”，就是在相同优先级的运算符中，先由运算符左边的操作数开始处理，再处理右边的操作数。在这个式子中，由于“/”与“*”的优先级相同，因此d会先除以5再乘以4，得到的结果如上b后，将整个值赋给a存放。

6.2 表达式

本节视频教学录像：36 分钟

表达式是由常量、变量或是其他操作数与运算符所组合而成的语句。如下面的例子，均是表达式正确的使用方法。

```
-49 // 表达式由一元运算符“-”与常量49组成
sum + 2 // 表达式由变量sum、算术运算符与常量2组成
a + b - c / (d * 3 - 9) // 表达式由变量、常量与运算符所组成
```

此外，Java 还有一些相当简洁的写法，是将算术运算符和赋值运算符结合成为新的运算符。下表列出了这些运算符。

运算符	范例用法	说明	意义
+=	a += b	a + b 的值存放到 a 中	a = a + b
-=	a -= b	a - b 的值存放到 a 中	a = a - b
*=	a *= b	a * b 的值存放到 a 中	a = a * b
/=	a /= b	a / b 的值存放到 a 中	a = a / b
%=	a %= b	a % b 的值存放到 a 中	a = a % b

下面的几个表达式，皆是简洁的写法。

```
a++ // 相当于 a = a + 1
a -= 5 // 相当于 a = a - 5
b %= c // 相当于 b = b % c
a /= b- // 相当于计算 a = a / b 之后，再计算 b--
```

这种独特的写法虽然看起来有些怪异，但是它却可以减少程序的行数，提高运行的速度！看下面这个范例。

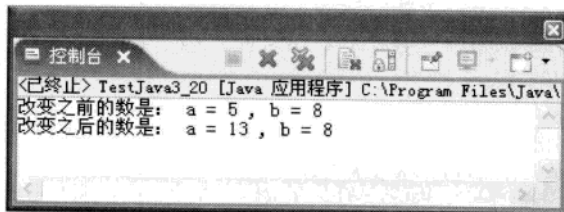
【范例 6-8】 程序的简洁写法（代码 6-8.java）。

```
01 // 下面是关于简洁写法的一段程序
```

```

02 public class TestJava3_20
03 {
04     public static void main(String[] args)
05     {
06         int a = 5, b = 8;
07
08         System.out.println("改变之前的数是: a = "+a+", b = "+b);
09         a += b;
10         System.out.println("改变之后的数是: a = "+a+", b = "+b);
11     }
12 }
    
```

程序运行结果如图所示。



【代码详解】

第 6 行分别把变量 a、b 赋值为 5 和 8。

第 8 行在运算之前先输出变量 a、b 的值，a 为 5，b 为 8。

第 9 行计算 a+=b，这个语句也就相当于 a=a+b，将 a+b 的值存放到 a 中。计算 5+8 的结果后赋值给 a 存放。

在第 10 行，再输出运算之后变量 a、b 的值，所以 a 的值变成 13，而 b 仍为 8。

下表列出了一些简洁写法的运算符及其范例说明。

运算符	范例	执行前		说明	执行后	
		a	b		a	b
+=	a += b	12	4	a + b 的值存放到 a 中 (同 a = a + b)	16	4
-=	a -= b	12	4	a - b 的值存放到 a 中 (同 a = a - b)	8	4
*=	a *= b	12	4	a * b 的值存放到 a 中 (同 a = a * b)	48	4
/=	a /= b	12	4	a / b 的值存放到 a 中 (同 a = a / b)	3	4
%=	a %= b	12	4	a % b 的值存放到 a 中 (同 a = a % b)	0	4
b++	a *= b++	12	4	a * b 的值存放到 a 后，b 加 1 (同 a = a * b; b++)	48	5
++b	a *= ++b	12	4	b 加 1 后，再将 a * b 的值存放到 a (同 b++; a = a * b)	60	5
b--	a *= b--	12	4	a * b 的值存放到 a 后，b 减 1 (同 a = a * b; b--)	48	3
--b	a *= --b	12	4	b 减 1 后，再将 a * b 的值存放到 a (同 b--; a = a * b)	36	3

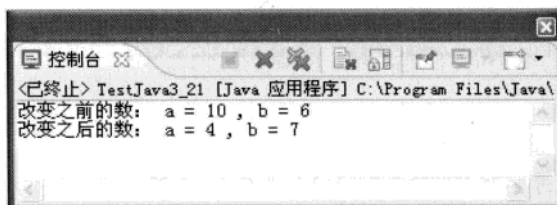
下面举一个实例来说明这些简洁的表达式在程序中该如何应用。以 TestJava3_21 为例，输入

两个数，经过运算之后，来看看这两个变量所存放的值有什么变化。

【范例 6-9】 程序的简洁写法 2（代码 6-9.java）。

```
01 // 下面的程序说明了简洁表达式的使用方法，但这种方式现在已不提倡使用了
02 public class TestJava3_21
03 {
04     public static void main(String[] args)
05     {
06         int a = 10, b = 6;
07
08         System.out.println("改变之前的数: a = "+a+", b = "+b);
09         a -= b++;           // 先计算 a-b 的值，将结果设给 a 之后，再将 b 值加 1
10         System.out.println("改变之后的数: a = "+a+", b = "+b);
11     }
12 }
```

程序运行结果如图所示。



【代码详解】

第 8 行输出运算前变量 a、b 的值。在程序中 a、b 的赋值为 10 和 6，因此输出的结果 a 为 10，b 为 6。

第 9 行计算 `a -= b++`，也就是执行下面这两条语句。

```
a = a - b;           // (a = 10 - 6 = 4, 所以 a = 4)
b++;                 // (b = b + 1 = 6 + 1 = 7, 所以 b = 7)
```

在第 10 行，将经过运算之后的结果输出，即可得到 a 为 4、b 为 7 的答案。

6.2.1 算术表达式

算术表达式用于数值计算，由算术运算符和变量或常量组成，其结果是一个数值。

【范例 6-10】 简单的算术表达式的使用（代码 6-10.java）。

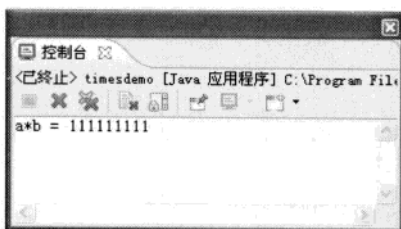
```
01 public class timesdemo
02 {
03     public static void main(String[] args)
```

```

04      {
05          int a = 12345679, b = 9;
06
07          System.out.println("a*b = "+a*b);
08      }
09  }

```

程序运行结果如图所示。



6.2.2 关系表达式

关系表达式常用于程序判断语句中，由关系运算符组成，其运算结果为逻辑型。

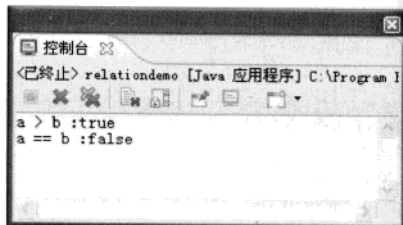
【范例 6-11】 简单的关系表达式的使用（代码 6-11.java）。

```

01  public class relationdemo {
02      public static void main(String[] args) {
03          int a = 5, b = 4;
04          boolean t1 = a > b;
05          boolean t2 = a == b;
06          System.out.println("a > b : " + t1);
07          System.out.println("a == b : " + t2);
08      }
09  }

```

程序运行结果如图所示。



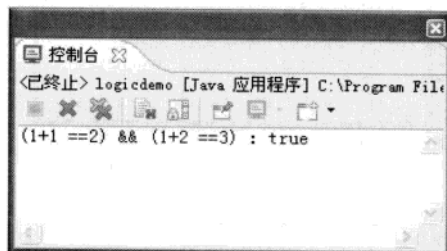
6.2.3 逻辑表达式

逻辑表达式是由逻辑运算符组成的表达式，其结果也为逻辑型。

【范例 6-12】 简单的逻辑表达式的使用（代码 6-12.java）。

```
01 public class logicdemo {
02     public static void main (String args[])
03     {
04         boolean t = (1+1 == 2) && (1+2 ==3);
05         System.out.println("(1+1 ==2) && (1+2 ==3) : "+ t);
06     }
07 }
```

程序运行结果如图所示。



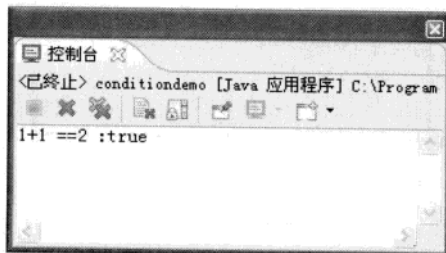
6.2.4 条件表达式

条件表达式由运算符“?:”组成，其基本形式为 a?x:y，如果 a 值为非 0，则整个表达式的值为 x，否则为 y。

【范例 6-13】 简单的条件表达式的使用（代码 6-13.java）。

```
01 public class conditiondemo {
02     public static void main (String args [])
03     {
04         boolean t = (1+1 == 2)? true : false;
05         System.out.println("1+1 ==2 :"+ t);
06     }
07 }
```

程序运行结果如下。



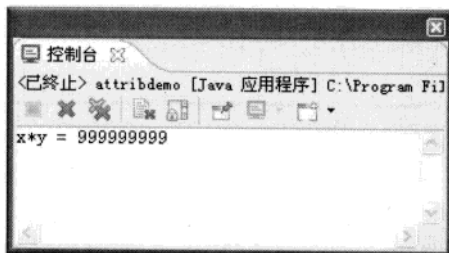
6.2.5 赋值表达式

赋值表达式由赋值运算符和操作数组成，赋值运算符用于给变量赋值。

【范例 6-14】 简单的赋值表达式的使用（代码 6-14.java）。

```
01 public class attribdemo {
02     public static void main (String args[])
03     {
04         int x=12345679 , y=81 ,z;
05         z = x*y;
06         System.out.println("x*y = " +z);
07     }
08 }
```

程序运行结果如图所示。



6.2.6 表达式的类型转换

当 int 类型遇上了 float 类型，到底谁是“赢家”呢？在前面曾提到过数据类型的转换，在这里，要再一次详细讨论表达式的类型转换。

Java 是一个很有弹性的程序设计语言，当上述情况发生时，只要坚持“以不流失数据为前提”的大原则，即可进行不同的类型转换，使不同类型的数据、表达式都能继续存储。依照大原则，当 Java 发现程序的表达式中有类型不相符的情况时，就会依据下列规则来处理类型的转换。

- (1) 占用字节较少的类型转换成占用字节较多的类型。
- (2) 字符类型会转换成 int 类型。
- (3) int 类型会转换成 float 类型。
- (4) 表达式中若某个操作数的类型为 double，则另一个操作数也会转换成 double 类型。
- (5) 布尔类型不能转换成其他类型。

【范例 6-15】 表达式类型的自动转换（代码 6-15.java）。

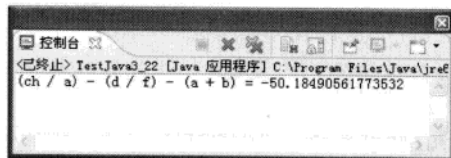
```
01 // 下面的程序说明了表达式类型的自动转换问题
02 public class TestJava3_22
03 {
```



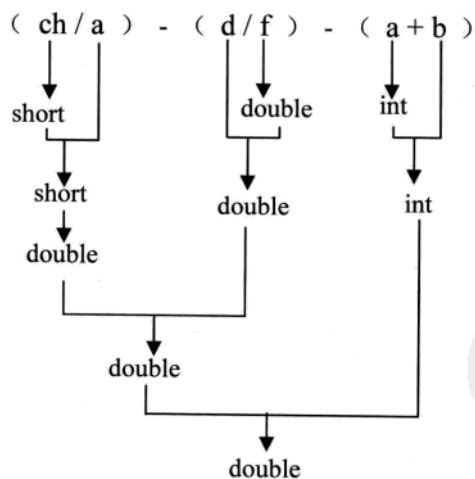
```

04      public static void main(String[] args)
05      {
06          char ch = 'a';
07          short a = -2;
08          int b = 3;
09          float f = 5.3f;
10          double d = 6.28;
11
12          System.out.print("(ch / a) - (d / f) - (a + b) = ");
13          System.out.println((ch / a) - (d / f) - (a + b));
14      }
15  }
    
```

程序运行结果如图所示。



先别急着看结果，在程序运行之前可先思考一下，这个复杂的表达式 $(ch / a) - (d / f) - (a + b)$ 最后的输出类型是什么？它又是如何将不同的数据类型转换成相同的呢？读者可以参考下图的分析过程。



6.3 语句

在学会使用运算符和表达式后，就可以写出最基本的 Java 程序语句了。表达式由运算符和操作数组成，语句则由表达式组成。例如 $a + b$ 是一个表达式，加上分号后就成为了下面的形式。

```
a + b ;
```

这就是一个语句。计算机执行程序就是一个语句一个语句进行的。

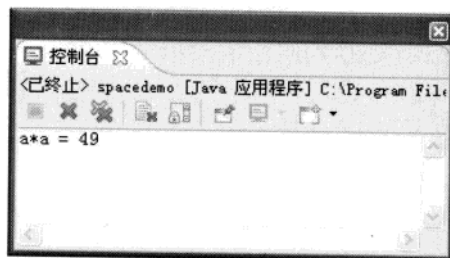
6.3.1 语句中的空格

在 Java 程序语句中，空格是必不可少的。空格可以使程序阅读起来更方便易懂。例如下面的程序使用了空格，程序的作用很容易看懂。

【范例 6-16】 语句中的空格使程序易懂（代码 6-16.java）。

```
01 public class spacedemo {  
02     public static void main (String args[])  
03     {  
04         int a;  
05         a=7;  
06         a=a*a;  
07         System.out.println("a*a = " + a);  
08     }  
09 }
```

程序运行结果如下。



6.3.2 空语句

前面所讲的语句都要进行一定的操作，但是 Java 中有一种语句什么也不执行，这就是空语句。

空语句是由一个分号组成的语句，空语句一般用于在调试时留空以待以后添加新的功能。如果不是出于这种目的，一般不建议使用空语句，因为空语句不完成任何功能，但同样会额外占用计算机资源。

6.3.3 声明语句

在前面已经多次用到了声明语句。其格式一般如下。

<声明数据类型> <变量 1> ...<变量 n>;

使用声明语句可以每一条语句声明一个变量，也可以在一条语句中声明多个变量。还可以在声明变量的同时，直接与赋值语句连用为变量赋值。例如：

```
int a;  
int x, y;  
int t = 1;
```

6.3.4 赋值语句

除了可以在声明语句中为变量赋初值，还可以在程序中使用赋值语句为变量重新赋值。例如：

```
pi = 3.1415;  
r = 25;  
s = pi*r*r;
```

在这个程序代码中，使用赋值语句给变量赋值，等号右边可以是一个常量或变量，也可以是一个表达式，这样程序在运行时先计算表达式的值，然后将结果赋给等号左边的变量。

6.4 练一练

一、填空题

1. 括号运算符用于处理表达式的_____。
2. 算术表达式的结果是_____。

二、简答题

简述在 Java 中，表达式类型转换的规则。

6.5 跟我上机

编写程序，计算表达式 “((12345679*9) > (97654321*3)) ? true : false” 的值。

第7章

改变程序执行方向——程序控制结构



本章视频教学录像：35 分钟

程序之所以能够按照人们的意愿执行，主要依靠的就是程序的控制结果。本章重点介绍选择与循环结构语句，学习如何利用这些不同的结构编写出有趣的程序，让程序的编写更灵活，操控更方便。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握程序结构的设计方法
- ☐ 掌握选择结构的类型与用法
- ☐ 掌握循环结构的类型与用法
- ☐ 掌握循环的跳离语句的运用方法



7.1 程序的结构设计

▶ 本节视频教学录像：1 分钟

结构化程序设计语言，强调用模块化、积木式来建立程序。采用结构化程序设计方法，可使程序的逻辑结构清晰、层次分明、可读性好、可靠性强，从而提高了程序的开发效率，保证了程序质量，改善了程序的可靠性。

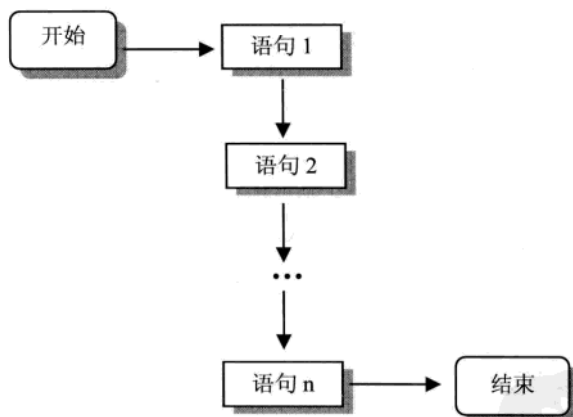
一般来说程序的结构包含以下 3 种。

- (1) 顺序结构
- (2) 选择结构
- (3) 循环结构

这 3 种不同的结构有一个共同点，就是它们都只有一个入口，也只有一个出口。程序中使用了上面这些结构到底有什么好处呢？这些单一入、出口可以让程序易读、好维护，也可以减少调试的时间。

7.1.1 顺序结构

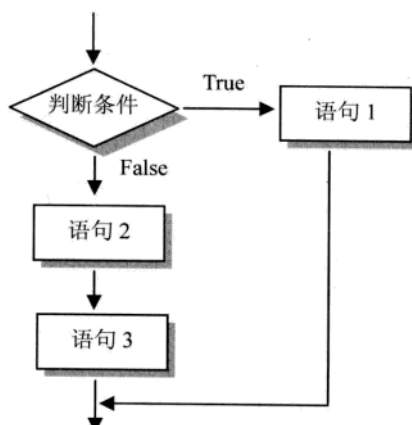
结构化程序的最简单的结构就是顺序结构，所谓顺序结构程序就是按书写顺序执行的语句构成的程序段，其流程如图所示。



通常情况下，程序的语句按顺序一句一句地执行，构成了顺序结构。有一些程序并不按顺序执行语句，这个过程称为“控制的转移”，它涉及了另外两类程序的控制结构，即分支结构和循环结构。

7.1.2 选择结构

选择结构也称为分支结构，在许多实际问题的程序设计中，根据输入数据和中间结果的不同情况需要选择不同的语句组执行，在这种情况下，必须根据某个变量或表达式的值作出判断，以决定执行哪些语句和跳过哪些语句不执行，其流程如图所示。



选择结构是根据给定的条件进行判断, 决定执行某个分支的程序段。条件分支主要用于两个分支的选择, 由 if 语句和 if... else 语句来实现。

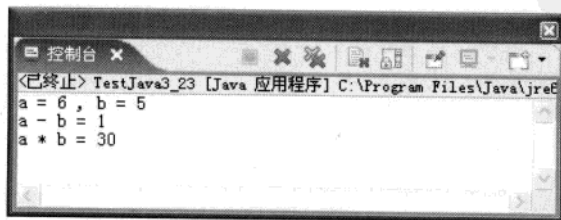
这种结构可以依据判断条件的结构, 来决定要执行的语句。当判断条件的值为真时, 就运行“语句 1”; 当判断条件的值为假时, 则执行“语句 2”。不论执行哪一个语句, 最后都会再回到“语句 3”继续执行。举例来说, 想在下面的程序中声明两个整数 a 及 b, 并赋其初值, 如果 a 大于 b, 在显示器中输出 a-b 的计算结果。无论 a 是否大于 b, 最后均输出 a*b 的值。

【范例 7-1】 if 语句的使用 (代码 7-1.java)。

```

01 // 下面的程序说明了 if 语句的操作, 只有当条件满足时才会被执行
02 public class TestJavaif
03 {
04     public static void main(String[] args)
05     {
06         int a = 6, b = 5;
07
08         System.out.println("a = "+a+", b = "+b);
09         if(a>b)
10             System.out.println("a - b = "+(a-b));
11         System.out.println("a * b = "+(a*b));
12     }
13 }
  
```

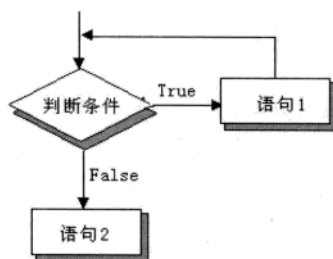
程序运行结果如下。



读者可以试着更改程序第 6 行中变量 a、b 的初值，将 a 的值设置得比 b 值小，可较容易观察程序运行的流程。如果输入的 a 值小于或等于 b 值，则会跳至第 11 行执行。

7.1.3 循环结构

循环结构是程序中的另一种重要结构，它和顺序结构、选择结构共同作为各种复杂程序的基本构造部件。循环结构的特点是在给定条件成立时，反复执行某个程序段。通常我们称给定条件为循环条件，称反复执行的程序段为循环体。循环体可以是复合语句、单个语句或空语句。在循环体中也可以包含循环语句，实现循环的嵌套。循环结构的流程如图所示。



7.2 选择结构

本节视频教学录像：12 分钟

Java 语言中的选择结构提供了以下两种类型的分支结构。

条件分支：根据给定的条件进行判断，决定执行某个分支的程序段。

开关分支：根据给定整型表达式的值进行判断，然后决定执行多路分支中的一支。

条件分支主要用于两个分支的选择，由 if 语句和 if ... else 语句来实现。开关分支用于多个分支的选择，由 switch 语句来实现。在语句中加上了选择结构之后，就像是十字路口，根据不同的选择，程序的运行会有不同的结果。

7.2.1 if 语句

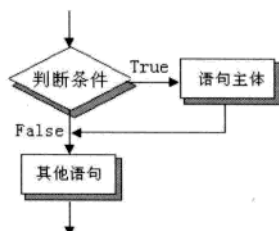
if 语句用于实现条件分支结构，它在可选动作中作出选择，执行某个分支的程序段。if 语句有两种格式在使用中供选择。要根据判断的结构来执行不同的语句时，使用 if 语句是一个很好的选择，它会准确地检测判断条件成立与否，再决定是否要执行后面的语句。

if 语句的格式如下。

```

if (判断条件){
    语句 1 ;
    ...
    语句 2 ;
}
    
```

若是在 if 语句主体中要处理的语句只有 1 个,可省略左、右大括号。当判断条件的值不为假时,就会逐一执行大括号里面所包含的语句。if 语句的流程如图所示。



选择结构中除了 if 语句之外,还有 if...else 语句。在 if 语句中如果判断条件成立,即可执行语句主体内的语句,但若要在判断条件不成立时可以执行其他的语句,使用 if...else 语句就可以节省判断的时间。

另外如果表达式的值为真(非零值),则执行 if 语句中的语句;否则当表达式的值为假(零值)时,将执行整个 if 语句下面的其他语句。if 语句中的语句可以是一条语句,也可以是复合语句。

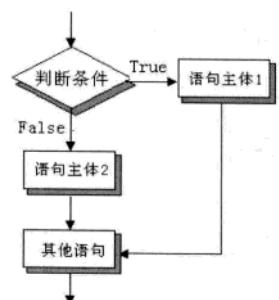
7.2.2 if...else 语句

if...else 语句是判断条件是否成立来执行的,如果条件表达式的值为真,则执行语句体 1;判断条件不成立时,则会执行 else 后面的语句主体 2,然后继续执行整个 if 语句后面的语句。语句体 1 和语句体 2 可以是一条语句,也可以是复合语句。if...else 语句的格式如下。

```

if (判断条件)
{
    语句主体 1 ;
}
else
{
    语句主体 2;
}
  
```

若是在 if 语句或 else 语句主体中要处理的语句只有一个,可以将左、右大括号去除。if...else 语句的流程如图所示。

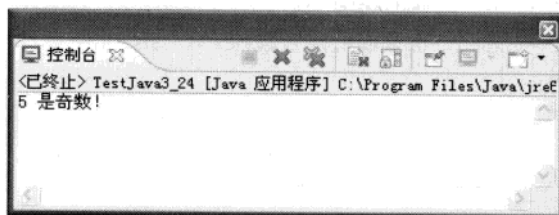


下面举一个简单的例子：声明一个整型变量 *a*，并对其赋初值 5，在程序中判断 *a* 是奇数还是偶数，再将判断的结果输出。

【范例 7-2】 if 语句的使用（代码 7-2.java）。

```
01 // 以下程序说明了 if...else 的使用方法
02 public class TestJavaif2
03 {
04     public static void main(String[] args)
05     {
06         int a = 5 ;
07
08         if(a%2 == 1)
09             System.out.println(a+" 是奇数！");
10         else
11             System.out.println(a+" 是偶数！");
12     }
13 }
```

程序运行结果如图所示。



【代码详解】

第 8~11 行为 if...else 语句。在第 8 行中，if 的判断条件为 $a \% 2 == 1$ ，当 *a* 除以 2 取余数，若得到的结果为 1，表示 *a* 为奇数，若 *a* 除以 2 取余数得到的结果为 0，*a* 则为偶数。

当 *a* 除以 2 取余数的结果为 1 时，即执行第 9 行的语句，输出“*a* 是奇数！”；否则执行第 11 行，输出“*a* 是偶数！”。

读者可以自行更改变量 *a* 的初值，再重复执行程序。

从上面的程序中可以发现，程序的缩进在这种选择结构中起着非常重要的作用，它可以使设计者编写的程序结构层次清晰，在维护上也就比较简单。建议读者以后在编写程序时要养成缩进的好习惯。

7.2.3 if...else if...else 语句

如果需要在 if...else 里判断多个条件，就需要使用 if...else if ... else 语句。其格式如下。

```
if (条件判断 1){
```

```

        语句主体 1 ;
    }else if (条件判断 2){
        语句主体 2 ;
    }
    ... // 多个 else if()语句
    else{
        语句主体 3 ;
    }

```

这种方式用在含有多个判断条件的程序中，请看下面的范例。

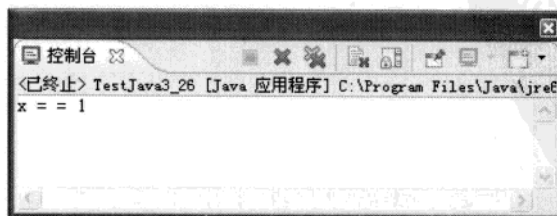
【范例 7-3】 多分支条件语句的使用（代码 7-3.java）。

```

01 // 以下程序说明了多分支条件语句 if...else if ...else 的使用
02 public class TestJavaduofenzhi
03 {
04     public static void main(String[] args)
05     {
06         int x = 1 ;
07
08         if(x= =1)
09             System.out.println("x = 1");
10         else if(x= =2)
11             System.out.println("x = 2");
12         else if(x= =3)
13             System.out.println("x = 3");
14         else
15             System.out.println("x > 3");
16     }
17 }

```

程序运行结果如图所示。



可以看出，if ... else if ...else 比单纯的 if...else 语句可以含有更多的条件判断语句。可是读者想一想，如果有很多条件都要判断的话，这样写会不会是一件很头疼的事情，下面介绍的多重选择语句就可以解决这一令人头疼的问题。

7.2.4 条件运算符

还有一种运算符可以代替 if...else 语句，即条件运算符，如表所示。

条件运算符	意义
?:	根据条件的成立与否，来决定结果为“:”前或“:”后的表达式

使用条件运算符时，操作数有 3 个，其格式如下。

条件判断? 表达式 1: 表达式 2

将上面的格式以 if 语句解释，就是当条件成立时执行表达式 1，否则执行表达式 2。通常会将这两个表达式之一的运算结果指定给某个变量，也就相当于下面的 if...else 语句。

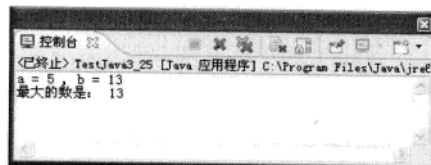
```
if (条件判断)
    变量 x = 表达式 1 ;
else
    变量 x = 表达式 2 ;
```

接下来，可以试着练习用条件运算符来编写程序。在下面的程序中声明变量 a、b，并对其赋初值，再利用条件运算符判断其最大值。

【范例 7-4】 条件运算符的使用（代码 7-4.java）。

```
01 // 以下程序说明了条件运算符的使用方法
02 public class TestJavatiaojian
03 {
04     public static void main(String[] args)
05     {
06         int a = 5, b = 13, max ;
07
08         max = (a>b)?a:b ;
09
10         System.out.println("a = "+a+", b = "+b);
11         System.out.println("最大的数是: "+max);
12     }
13 }
```

程序运行结果如图所示。



【代码详解】

第 6 行声明变量并对其赋初值。a、b 为要比较其大小的两个整数值，max 存放比较大小后的最大的那个值。

第 8 行 ($\text{max} = (\text{a} > \text{b}) ? \text{a} : \text{b}$) 赋值当 $\text{a} > \text{b}$ 时， $\text{max} = \text{a}$ ，否则 $\text{max} = \text{b}$ 。

第 10 行输出 a、b 的值，第 11 行输出最大值。

可以自行将 a、b 的值更改，再运行此程序。

读者可以发现，使用条件运算符编写程序时较为简洁，它用一个语句就可以替代一长串的 if...else 语句，所以条件运算符的执行速度也较高。

7.2.5 多重选择——switch 语句

对于多路分支处理还有一种情况，它对一个整型表达式的值进行计算，针对该表达式的不同取值决定执行多路分支程序段中的一支，这就是 switch 结构。使用嵌套 if...else 语句最常发生的状况，就是容易将 if 与 else 配对混淆而造成阅读及运行上的错误。使用 switch 语句则可避免这种错误的发生。

switch 语句的格式如下。

```
switch (表达式)
{
    case 选择值 1 : 语句主体 1 ;
    break ;
    case 选择值 2 : 语句主体 2 ;
    break ;
    .....
    case 选择值 n : 语句主体 n ;
    break ;
    default: 语句主体 ;
}
```

switch 结构也称为“多路选择结构”，它在许多不同的语句组之间作出选择。switch 语句用于实现该结构，它常与 break 语句联合使用，break 语句用于转换程序的流程，在 switch 语句中使用 break 语句可以使程序立即退出该结构，转而执行该结构后面的第 1 条语句。



提示：需要特别注意的是，switch 语句里的选择值只能是字符或者常量。

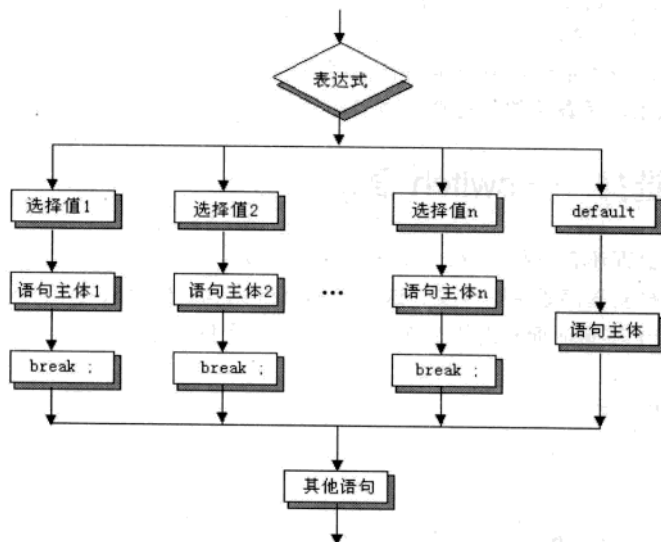
接下来看看 switch 语句执行的流程。

- (1) switch 语句先计算括号中表达式的结果。
- (2) 根据表达式的值检测是否符合执行 case 后面的选择值，若是所有 case 的选择值皆不符合，则执行 default 所包含的语句，执行完毕即离开 switch 语句。
- (3) 如果某个 case 的选择值符合表达式的结果，就会执行该 case 所包含的语句，直到遇到

break 语句后才离开 switch 语句。

(4) 若是没有 case 语句结尾处加上 break 语句，则会一直执行到 switch 语句的尾端才会离开 switch 语句。break 语句在下面的章节中会介绍，读者只要先记住 break 是跳出语句就可以了。

(5) 若是没有定义 default 该执行的语句，则什么也不会执行，而是直接离开 switch 语句。根据上面的描述，可以绘制出如图所示的 switch 语句流程。



下面的程序是一个简单的赋值表达式，利用 switch 语句处理此表达式中的运算符，再输出运算后的结果。

【范例 7-5】 多分支条件语句的使用（代码 7-5.java）。

```

01 // 以下程序说明了多分支条件语句的使用
02 public class TestJavaswitch
03 {
04     public static void main(String[] args)
05     {
06         int a = 100, b = 7;
07         char oper = '+';
08
09         switch(oper)                // 用 switch 实现多分支语句
10         {
11             case '+':
12                 System.out.println(a + " + " + b + " = " + (a+b));
13                 break;
14             case '-':
15                 System.out.println(a + " - " + b + " = " + (a-b));
16                 break;

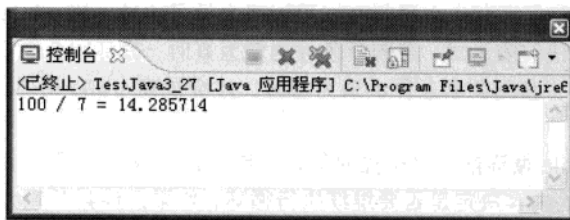
```

```

17         case '*':
18             System.out.println(a+" * "+b+" = "+(a*b));
19             break ;
20         case '/':
21             System.out.println(a+" / "+b+" = "+((float)a/b));
22             break ;
23         default:
24             System.out.println("未知的操作！");
25     }
26 }
27 }
28 }

```

程序运行结果如图所示。



【代码详解】

第 7 行，利用变量存放一个运算符，如 3+2、5*7 等。

第 9~25 行为 switch 语句。当 oper 为字符+、-、*、/、%时，输出运算的结果后离开 switch 语句；若所输入的运算符皆不是这些，即执行 default 所包含的语句，输出“未知的操作！”，再离开 switch。

选择值为字符时，必须用单引号将字符包围起来。

程序运行的结果会因为没有加上 break 语句而出现错误，所以程序设计者在使用 switch 语句的时候，要特别注意是否需要加上 break 语句。

7.3 循环结构

本节视频教学录像：8 分钟

循环结构是程序中的另一种重要结构，它和顺序结构、选择结构共同作为各种复杂程序的基本构造部件。循环结构的特点是在给定条件成立时，反复执行某个程序段。通常我们称给定条件为循环条件，称反复执行的程序段为循环体。循环体可以是复合语句、单个语句或空语句。在循环体中也可以包含循环语句，实现循环的嵌套。

循环结构包括 while 循环、for 循环，还可以使用嵌套循环完成复杂的程序控制操作。

7.3.1 while 循环

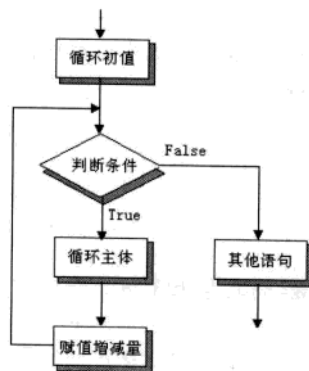
while 循环语句的执行过程是先计算表达式的值，若表达式的值为真（非零），则执行循环体中的语句，继续循环；否则退出该循环，执行 while 语句后面的语句。循环体可以是一条语句或空语句，也可以是复合语句。while 循环的格式如下。

```
while (判断条件)
{
    语句 1 ;
    语句 2 ;
    ...
    语句 n ;
}
```

当 while 循环主体有且只有一个语句时，可以将大括号去掉。在 while 循环语句中，只有一个判断条件，它可以是任何表达式，当判断条件的值为真时，循环就会执行一次，再重复测试判断条件、执行循环主体，直到判断条件的值为假时，才会跳离 while 循环。下面列出了 while 循环执行的流程。

- (1) 第 1 次进入 while 循环前，必须先对循环控制变量（或表达式）赋起始值。
- (2) 根据判断条件的内容决定是否要继续执行循环，如果条件判断值为真（True），则继续执行循环主体。
- (3) 条件判断值为假（False），则跳出循环执行其他语句。
- (4) 执行完循环主体内的语句后，重新对循环控制变量（或表达式）赋值（增加或减少）。由于 while 循环不会自动更改循环控制变量（或表达式）的内容，所以在 while 循环中对循环控制变量赋值的工作要由设计者自己来做，完成后再回到步骤 2 重新判断是否继续执行循环。

while 循环流程如图所示。



下面这个范例是循环计算 1 累加至 10。

【范例 7-6】 while 循环的使用（代码 7-6.java）。

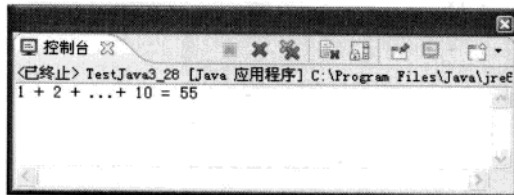
```
01 // 以下程序说明了 while 循环的使用方法
```

```

02 public class TestJavawhile
03 {
04     public static void main(String[] args)
05     {
06         int i = 1, sum = 0;
07
08         while(i<=10)
09         {
10             sum += i;                // 累加计算
11             i++;
12         }
13         System.out.println("1 + 2 + ... + 10 = "+sum);    // 输出结果
14     }
15 }

```

程序运行结果如图所示。



【代码详解】

在第 6 行中，将循环控制变量 *i* 的值赋值为 1。

第 8 行进入 `while` 循环的判断条件为 `i<=10`。第 1 次进入循环时，由于 *i* 的值为 1，所以判断条件的值为真，即进入循环主体。

第 9~12 行为循环主体，`sum+i` 后再指定给 `sum` 存放，*i* 的值加 1，再回到循环起始处，继续判断 *i* 的值是否仍在所限定的范围内，直到 *i* 大于 10 即跳出循环，表示累加的操作已经完成，最后再将 `sum` 的值输出即可。

7.3.2 do...while 循环

在 `do...while` 循环中，首先判定循环的条件，若循环的条件不满足，循环体则一次也不执行。直到型循环与当型循环不同，它首先执行一次循环体，然后判定循环条件。`do...while` 循环的格式如下。

```

do{
    语句 1 ;
    语句 2 ;
    .....
    语句 n ;
}

```


}while (判断条件);

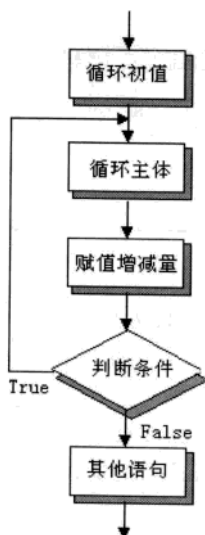
do...while 循环的执行过程是先执行一次循环体，然后判断表达式的值，如果是真（非零），则再执行循环体，继续循环；否则退出循环，执行下面的语句。循环体可以是单条语句或是复合语句，在语法上它也可以是空语句，但此时循环没有什么实际意义。下面列出 do...while 循环执行的流程。

(1) 在进入 do...while 循环前，要先对循环控制变量（或表达式）赋起始值。

(2) 直接执行循环主体，循环主体执行完毕，才开始根据判断条件的内容决定是否继续执行循环：条件判断值为真（True）时，继续执行循环主体；条件判断值为假（False）时，则跳出循环，执行其他语句。

(3) 执行完循环主体内的语句后，重新对循环控制变量（或表达式）赋值（增加或减少）。由于 do...while 循环和 while 循环一样，不会自动更改循环控制变量（或表达式）的内容，所以在 do...while 循环中赋值循环控制变量的工作要由自己来做，再回到步骤 2 重新判断是否继续执行循环。

do...while 循环流程如图所示。



把 TestJava3_28.java (1+2+...+10) 的程序稍加修改，用 do...while 循环设计一个能累加至 n 的程序，并且能够限制 n 的范围（n 要大于 0），就是下面的范例 TestJava3_29.java。

【范例 7-7】 do...while 循环语句的使用（代码 7-7.java）。

```

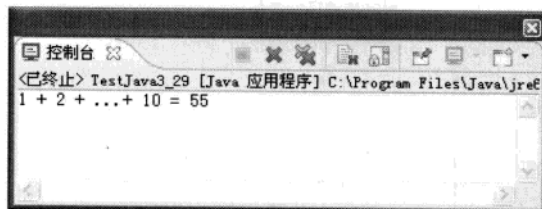
01 // 以下程序说明了 do...while 循环的使用
02 public class TestJavadowhile
03 {
04     public static void main(String[] args)
05     {
06         int i = 1, sum = 0;
07         // do...while 是先执行一次，再进行判断，即循环体至少会被执行一次
    
```

```

08      do
09      {
10          sum += i;                // 累加计算
11          i++;
12      }while(i<=10);
13      System.out.println("1 + 2 + ... + 10 = "+sum);    // 输出结果
14  }
15  }

```

程序运行结果如图所示。



首先, 声明程序中要使用的变量 i (循环记数及累加操作数) 及 sum (累加的总和), 并将 sum 设初值为 0; 由于要计算 $1+2+\dots+10$, 因此在第 1 次进入循环的时候, 将 i 的值设为 1, 接着判断 i 是否小于等于 10, 如果 i 小于等于 10, 则计算 $sum+i$ 的值后再指定给 sum 存放。 i 的值已经不能满足循环条件时, i 即会跳出循环, 表示累加的操作已经完成, 再输出 sum 的值, 程序即结束运行。

【代码注解】

第 8~12 行利用 `do...while` 循环计算 1~10 的数累加。

第 13 行输出 1~10 的数的累加结果: $1 + 2 + \dots + 10 = 55$ 。

`do...while` 循环不管条件是什么, 都是先做再说, 因此循环的主体最少会被执行一次。在日常生活中, 如果能够多加注意, 并不难找到 `do...while` 循环的影子! 举例来说, 在利用提款机提款前, 会先进入输入密码的画面, 让使用者输入 3 次密码, 如果皆输入错误, 即会将银行卡吞掉, 其程序的流程就是利用 `do...while` 循环设计而成的。

7.3.3 for 循环

在 `for` 循环中, 语句 1、2、3 可以都有, 也可以都没有, 还可以有其中的某一个或某两个。循环体可以是一条语句或空语句, 也可以是复合语句。当很明确地知道循环要执行的次数时, 就可以使用 `for` 循环, 其语句格式如下。

```

for (赋值初值; 判断条件; 赋值增减量)
{
    语句 1 ;
    ...
    语句 n ;
}

```

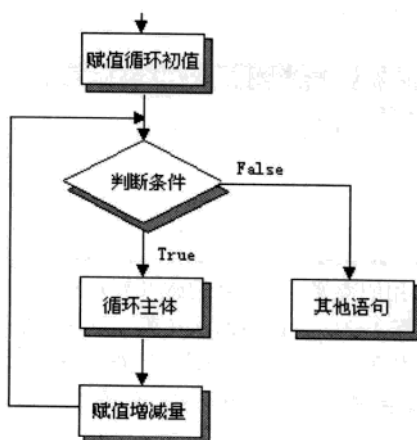
}

若是在循环主体中要处理的语句只有 1 个，可以将大括号去掉。下面列出 for 循环的流程。

(1) 第 1 次进入 for 循环时，对循环控制变量赋起始值。
 (2) 根据判断条件的内容检查是否要继续执行循环，当判断条件值为真（true）时，继续执行循环主体内的语句；判断条件值为假（false）时，则会跳出循环，执行其他语句。

(3) 执行完循环主体内的语句后，循环控制变量会根据增减量的要求，更改循环控制变量的值，再回到步骤 2 重新判断是否继续执行循环。

for 循环流程如图所示。



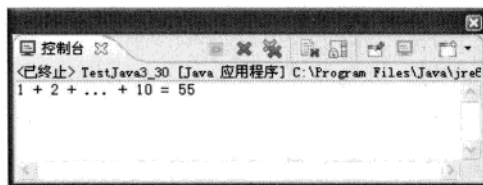
通过范例 TestJava3_30，读者应熟悉 for 循环的使用方法，它是利用 for 循环来完成由 1 至 10 的数的累加运算。

【范例 7-8】 for 循环的使用（代码 7-8.java）。

```

01 // 以下程序说明了 for 循环的使用方法
02 public class TestJavafor
03 {
04     public static void main(String[] args)
05     {
06         int i, sum = 0;
07         // for 循环的使用，用来计算数字累加之和
08         for(i=1;i<=10;i++)
09             sum += i;           // 计算 sum = sum+i
10         System.out.println("1 + 2 + ... + 10 = "+sum);
11     }
12 }
    
```

程序运行结果如图所示。



【代码详解】

第6行声明两个变量 `sum` 和 `i`, `i` 用于循环的计数控制。

第8~9行做 1~10 之间的循环累加, 执行的结果如 TestJava3_28 所示。如果读者不明白的话, 可以和 TestJava3_28 的程序说明比较一下, 相信就可以明白 `for` 的用法了。

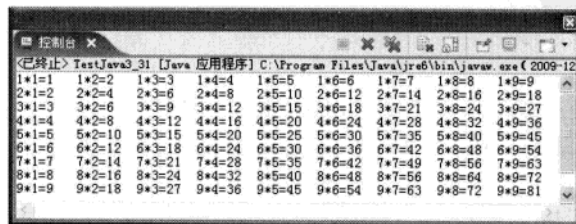
7.3.4 循环嵌套

当循环语句中又出现循环语句时, 就称为嵌套循环, 如嵌套 `for` 循环、嵌套 `while` 循环等。当然读者也可以使用混合嵌套循环, 也就是循环中又有其他不同种类的循环。下面以打印九九乘法表为例, 练习嵌套循环的用法。

【范例 7-9】 `for` 循环嵌套的使用 (代码 7-9.java)。

```
01 // 以下程序说明了 for 循环嵌套的使用方法
02 public class TestJavafor2
03 {
04     public static void main(String[] args)
05     {
06         int i, j;
07         // 用两层 for 循环输出乘法表
08         for(i=1; i<=9; i++)
09         {
10             for(j=1; j<=9; j++)
11                 System.out.print(i+"*"+j+"="+i*j+"\t");
12             System.out.print("\n");
13         }
14     }
15 }
```

程序运行结果如图所示。



【代码详解】

i 为外层循环的循环控制变量, j 为内层循环的循环控制变量。

当 i 为 1 时, 符合外层 for 循环的判断条件 ($i \leq 9$), 进入另一个内层 for 循环主体; 由于是第 1 次进入内层循环, 所以 j 的初值为 1, 符合内层 for 循环的判断条件 ($j \leq 9$), 进入循环主体, 输出 $i*j$ 的值 ($1*1=1$), j 再加 1 等于 2, 仍符合内层 for 循环的判断条件 ($j \leq 9$), 再次执行计算与输出的工作, 直到 j 的值大于 9 即离开内层 for 循环, 回到外层循环。此时, i 会加 1 成为 2, 符合外层 for 循环的判断条件, 继续执行内层 for 循环主体, 直到 i 的值大于 9 时即离开嵌套循环。

整个程序到底执行了几次循环? 可以看到, 当 i 为 1 时, 内层循环会执行 9 次 (j 为 1~9), 当 i 为 2 时, 内层循环也会执行 9 次 (j 为 1~9), 依次类推的结果, 这个程序会执行 81 次循环, 而显示器上也正好输出 81 个式子。

7.4 循环的跳转

▶ 本节视频教学录像: 14 分钟

在 Java 语言中, 有一些跳离的语句, 如 `break`、`continue` 等语句。`break` 语句和 `continue` 语句都是用来控制程序的流程转向的, 适当地和灵活地使用它们可以更方便或更简洁地进行程序的设计。

7.4.1 break 语句

在 `while`、`for`、`do...while` 或 `switch` 等语句结构中的循环体或语句组中使用 `break` 语句可以使程序立即退出该结构, 转而执行该结构下面的第 1 条语句。`break` 语句也称之为中断语句, 它通常用来在适当的时候退出某个循环, 或终止某个 `case` 并跳出 `switch` 结构。例如下面的 `for` 循环, 在循环主体中有 `break` 语句时, 当程序执行到 `break`, 即会离开循环主体, 而继续执行循环外层的语句。

```
for (初值赋值; 判断条件; 设增减量)
{
    语句 1 ;
    语句 2 ;
    ...
    break ;
    ...
    语句 n ;
}
```

// 若执行 `break` 语句, 则此块内的语句将不会被执行

以下的程序为例, 利用 `for` 循环输出循环变量 i 的值, 当 i 除以 3 所取的余数为 0 时, 即使用 `break` 语句的跳离循环, 并于程序结束前输出循环变量 i 的最终值。

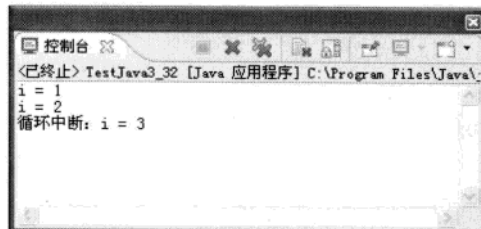
【范例 7-10】 break 语句的使用（代码 7-10.java）。

```

01 // 下面的程序是 break 的使用方法
02 public class TestJavabreak
03 {
04     public static void main(String[] args)
05     {
06         int i;
07
08         for(i=1;i<=10;i++)
09         {
10             if(i%3 == 0)
11                 break ;           // 跳出整个循环体
12             System.out.println("i = "+i);
13         }
14         System.out.println("循环中断: i = "+i);
15     }
16 }

```

程序运行结果如图所示。

**【代码详解】**

第 9~13 行为循环主体，i 为循环的控制变量。

当 $i\%3$ 为 0 时，符合 if 的条件判断，即执行第 11 行的 break 语句，跳离整个 for 循环。此例中，当 i 的值为 3 时， $3\%3$ 的余数为 0，符合 if 的条件判断，离开 for 循环，执行第 14 行：输出循环结束时循环控制变量 i 的值 3。

通常设计者都会设定一个条件，当条件成立时，不再继续执行循环主体。所以在循环中出现 break 语句时，if 语句通常也会同时出现。此外，读者可以回想一下前面提到过的 switch 语句中是不是也有一个 break，如果记不清楚可以看一下本章前面的内容。

7.4.2 continue 语句

在 while 和 do...while 语句的循环体中，执行 continue 语句将结束本次循环而立即测试循环的条件，以决定是否进行下一次循环。例如下面的 for 循环，在循环主体中有 continue 语句，当程序执行到 continue，即会回到循环的起点，继续执行循环主体的部分语句。

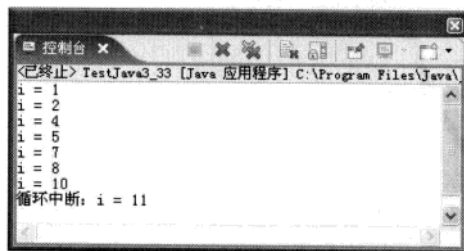
```
for (初值赋值; 判断条件; 设增减量)
{
    语句 1 ;
    语句 2 ;
    ...
    continue
    ...
    // 若执行 continue 语句, 则此处将不会被执行
    语句 n;
}
```

将程序 TestJava3_32 中的 break 语句改成 continue 语句, 就形成了程序 TestJava3_33.java。读者可以观察一下这两种跳离语句的不同之处。break 语句是跳离当前层循环, 而 continue 语句则是回到循环的起点。

【范例 7-11】 continue 语句的使用 (代码 7-11.java)。

```
01 // 下面的程序是 continue 的使用方法
02 public class TestJava continue
03 {
04     public static void main(String[] args)
05     {
06         int i ;
07
08         for(i=1;i<=10;i++)
09         {
10             if(i%3==0)
11                 continue ; // 跳出一次循环
12             System.out.println("i = "+i);
13         }
14         System.out.println("循环中断: i = "+i);
15     }
16 }
```

程序运行结果如图所示。



【代码注释】

第 9~13 行为循环主体， i 为循环控制变量。

当 $i \% 3$ 为 0 时，符合 if 的条件判断，即执行第 11 行的 continue 语句，跳离目前的 for 循环（不再执行循环体内的其他语句），而是回到循环开始处继续判断是否执行循环。此例中，当 i 的值为 3、6、9 时，取余数为 0，符合 if 判断条件，离开当前层的 for 循环，回到循环开始处继续判断是否执行循环。

当 i 的值为 11 时，不符合循环执行的条件，此时执行程序第 14 行，输出循环结束时循环控制变量 i 的值 11。

当判断条件成立时，break 语句与 continue 语句会有不同的执行方式。break 语句不管情况如何，先离开循环再说；而 continue 语句则不再执行此次循环的剩余语句，而是直接回到循环的起始处。

7.5 练一练

一、填空题

1. 在 Java 中，3 种基本的程序控制结构是顺序结构、_____和_____。
2. if 条件语句的 3 种形式为_____、_____和_____。

二、简答题

简述在 Java 中跳离循环语句的两种方法。

7.6 跟我上机

编写程序，使用循环控制语句计算“ $1+2+3+\cdots+100$ ”的值。

第 8 章

常用的数据结构——数组



本章视频教学录像：1 小时 2 分钟

数组是Java中一种常见的数据结构，分为一维数组、二维数组以及多维数组。只有灵活掌握数组的应用，才能编写出更强大、效率更高的Java程序。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握一维数组的使用
- ☐ 掌握二维数组的使用
- ☐ 了解数组越界的风险
- ☐ 熟悉多维数组的使用



若想存放一连串相关的数据，使用数组是个相当好的选择。

数组是由一组相同类型的变量所组成的数据类型，它们以一个共同的名称表示，数组中的个别元素则以标注来表示其存放的位置。数组依照存放元素的复杂程度，可分为一维数组、二维数组和多维数组等几种。

8.1 一维数组

▶ 本节视频教学录像：52 分钟

数组是有序数据的集合，数组中的每个元素具有相同的数据类型，可以用一个统一的数组名和下标来唯一地确定数组中的元素。一维数组可以存放上千万个数据，并且这些数据的类型是完全相同的。

8.1.1 一维数组的声明与内存的分配

要使用 Java 的数组，必须经过以下两个步骤。

- (1) 声明数组。
- (2) 分配内存给该数组。

这两个步骤的语法如下。

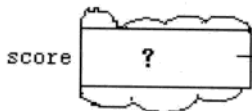
```
数据类型  数组名[] ;           //声明一维数组
数组名 = new 数据类型[个数];   //分配内存给数组
```

在数组的声明格式里，“数据类型”是声明数组元素的数据类型，常见的类型有整型、浮点型与字符型等。“数组名”是用来统一这组相同数据类型的元素的名称，其命名规则和变量相同，建议读者使用有意义的名称为数组命名。数组声明后，接下来便要配置数组所需的内存，其中“个数”是告诉编译器，所声明的数组要存放多少个元素，而“new”则是命令编译器根据括号里的个数，在内存中开辟一块内存供该数组使用。下面是关于一维数组的声明并分配内存给该数组的一个例子。

```
int score[] ;           //声明整型数组 score
score = new int[3];      //为整型数组 score 分配内存空间，其元素个数为 3
```

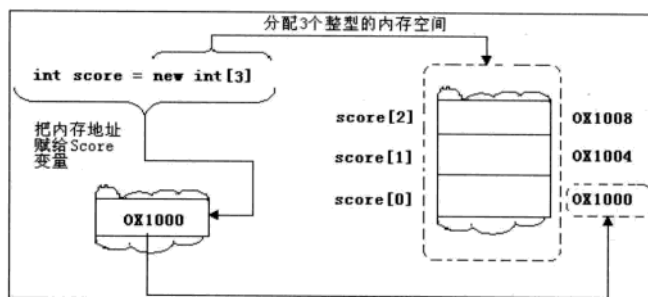
例子中的第 1 行，当声明一个整型数组 `score` 时，可将 `score` 视为数组类型的变量，此时这个变量并没有包含任何内容，编译器仅会分配一块内存给它，用来保存指向数组实体的地址，如图所示。

```
int score[]:
```



score 尚未指向数组实体的地址，所以 score 的内容未知

声明之后，接着要进行内存分配的操作，也就是例子中的第 2 行语句。这一行会开辟 3 个可供保存整数的内存空间，并把此内存空间的参考地址赋给 `score` 变量。其内存分配的流程如图所示。



图中的内存参考地址 0X1000 是假赋值，此值会因环境的不同而异。如第 3 章所述，数组是属于非基本数据类型，因此数组变量 `score` 所保存的并非是数组的实体，而是数组实体的参考地址。

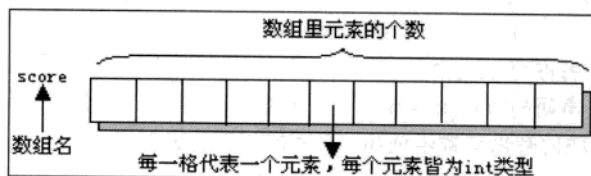
除了用两行来声明并分配内存给数组之外，也可以用较为简洁的方式，把两行缩成一行来编写，其格式如下。

```
数据类型 数组名[] = new 数据类型[个数]
```

上述的格式会在声明的同时，即分配一块内存空间供该数组使用。下面的例子是声明整型数组 `score`，并开辟可以保存 11 个整数的内存给 `score` 变量。

```
int score[] = new int[11]; // 声明一个元素个数为 11 的整型数组 score，同时开辟一块内存空间供其使用
```

在 Java 中，由于整数数据类型所占用的空间为 4 个 bytes，而整型数组 `score` 可保存的元素有 11 个，所以例子中占用的内存共有 $4 * 11 = 44$ 个字节。下图是将数组 `score` 用图来表示，从中可以比较容易理解数组的保存方式。



8.1.2 数组中元素的表示方法

想要使用数组里的元素，可以利用索引来完成。Java 的数组索引编号从 0 开始，以一个 `score[10]` 的整形数组为例，`score[0]` 代表第 1 个元素，`score[1]` 代表第 2 个元素，`score[9]` 为数组中的第 10 个元素（也就是最后一个元素）。下图为 `score` 数组中元素的表示及排列方式。



接下来看一个范例。在下面的程序里声明了一个一维数组，其长度为 3，利用 `for` 循环输出数组的内容后，再输出数组的元素个数。

【范例 8-1】 一维数组的使用（代码 8-1.java）。

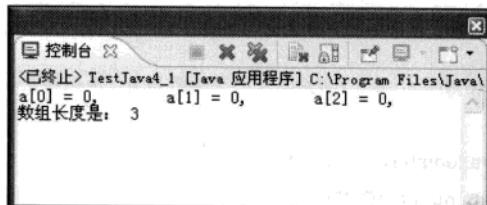
```

01 //下面这段程序说明了一维数组的使用方法
02 public class TestJavayiwei
03 {
04     public static void main(String args[])
05     {
06         int i;
07         int a[];                //声明一个整型数组 a
08         a=new int[3];           //开辟内存空间供整型数组 a 使用,其元素个数为 3
09
10         for(i=0;i<3;i++)        //输出数组的内容
11             System.out.print("a["+i+"] = "+a[i]+" ");
12
13         System.out.println("\n 数组长度是: "+a.length); //输出数组长度
14     }
15 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 7 行声明整型数组 a；第 8 行开辟了一块内存空间，以供整型数组 a 使用，其元素个数为 3。

第 10~11 行，利用 for 循环输出数组的内容。由于程序中并未对数组元素赋值，因此输出的结果都是 0。

第 13 行输出数组的长度。此例中数组的长度是 3，即代表数组元素的个数为 3。

需要特别注意的是，在 Java 中取得数组的长度（也就是数组元素的个数）可以利用“.length”完成，如下面的格式。

```
数组名.length
```

也就是说，若要取得 TestJava4_1 中所声明的数组 a 的元素个数，只要在数组 a 的名称后面加上“.length”即可，如下面的程序片段。

```
a.length; //取得数组的长度
```


8.1.3 数组初值的赋值

如果想直接在声明时就对数组赋初值，可以利用大括号完成。只要在数组的声明格式后面再加上初值的赋值即可，如下面的格式。

```
数据类型 数组名[] = {初值 0, 初值 1, ..., 初值 n}
```

大括号内的初值会依序指定给数组的第 1、…、n+1 个元素。此外，在声明的时候，并不需要将数组元素的个数列出，编译器会根据给出的初值个数来判断数组的长度。如下面的数组声明及赋初值例子。

```
int day[] = {32,23,45,22,13,45,78,96,43,32}; // 数组声明并赋初值
```

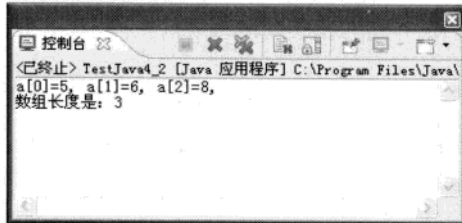
语句中声明了一个整型数组 day，虽然没有特别指明数组的长度，但是由于大括号里的初值有 10 个，所以编译器会分别依序指定给各元素存放，day[0]为 32，…，day[9]为 32。

【范例 8-2】 一维数组的赋值（代码 8-2.java）。

```
01 //一维数组的赋值，这里采用静态方式赋值
02 public class TestJavayiweifuzhi
03 {
04     public static void main(String args[])
05     {
06         int i;
07         int a[]={5,6,8}           //声明一个整数数组 a，并赋初值
08
09         for(i=0;i<a.length;i++) // 输出数组的内容
10             System.out.print("a["+i+"]="+a[i]+" ");
11
12         System.out.println("\n 数组长度是: "+a.length);
13     }
14 }
```

【运行结果】

保存并运行程序，结果如图所示。



除了在声明时就赋初值之外，也可以在程序中为某个特定的数组元素赋值。可以将程序

TestJava4_2 的第 7 行更改成下面的程序片段。

```
int a [] = new int[] ;
a[0] = 5 ;
a[1] = 6 ;
a[2] = 8 ;
```

8.1.4 数组应用范例

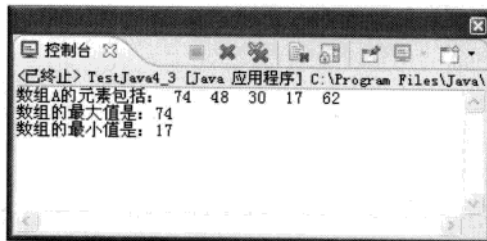
从前面的范例可知，数组的索引就好像饭店房间的编号一样，想要找到某个房间，就得先找到房间编号！接下来再举一个例子，说明如何将数组里的最大值及最小值列出。

【范例 8-3】 求数组中的最大值和最小值（代码 8-3.java）。

```
01 // 这个程序主要是求得数组中的最大值和最小值
02 public class TestJavazuidazuixiao
03 {
04     public static void main(String args[])
05     {
06         int i,min,max;
07         int A[]={74,48,30,17,62};           // 声明整数数组 A,并赋初值
08
09         min=max=A[0];
10         System.out.print("数组 A 的元素包括: ");
11         for(i=0;i<A.length;i++)
12         {
13             System.out.print(A[i]+" ");
14             if(A[i]>max)           // 判断最大值
15                 max=A[i];
16             if(A[i]<min)           // 判断最小值
17                 min=A[i];
18         }
19         System.out.println("\n 数组的最大值是: "+max); // 输出最大值
20         System.out.println("数组的最小值是: "+min);    // 输出最小值
21     }
22 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 6 行声明整数变量 `i` 作为循环控制变量及数组的索引，另外也声明了存放最小值的变量 `min` 与最大值的变量 `max`。

第 7 行声明整型数组 `A`，其数组元素有 5 个，其值分别为 74、48、30、17、62。

第 9 行将 `min` 与 `max` 的初值设为数组的第 1 个元素。

第 10~18 行逐一输出数组里的内容，并判断数组里的最大值与最小值。

第 19~20 行输出比较后的最大值与最小值。

将变量 `min` 与 `max` 初值设成数组的第 1 个元素后，再逐一与数组中的各元素相比。比 `min` 小，就将该元素的值指定给 `min` 存放，使 `min` 的内容保持最小；同样，当该元素比 `max` 大时，就将该元素的值指定给 `max` 存放，使 `max` 的内容保持最大。`for` 循环执行完成，也就表示数组中所有的元素都已经比较完毕，此时变量 `min` 与 `max` 的内容就是最小值与最大值。

8.1.5 与数组操作有关的 API 方法

在 Java 语言中提供了很多的 API 方法，供开发人员使用。下面介绍两种常用的数组操作方法，一个是数组的拷贝操作，另一个是数组的排序操作。其他的操作请读者自行查阅 JDK 帮助文档。

【范例 8-4】 数组的拷贝操作（代码 8-4.java）。

```
01 // 以下这段程序说明数组的拷贝操作
02 public class TestJavashuzucopy
03 {
04     public static void main(String[] args)
05     {
06         int a1[] = {1,2,3,4,5};           //声明两个整型数组 a1、a2，并进行静态初始化
07         int a2[] = {9,8,7,6,5,4,3};
08         System.arraycopy(a1,0,a2,0,3);    // 进行数组拷贝的操作
09         System.out.print("a1 数组中的内容: ");
10         for(int i=0;i<a1.length;i++)      // 输出 a1 数组中的内容
11             System.out.print(a1[i]+" ");
12         System.out.println();
13
14         System.out.print("a2 数组中的内容: ");
```

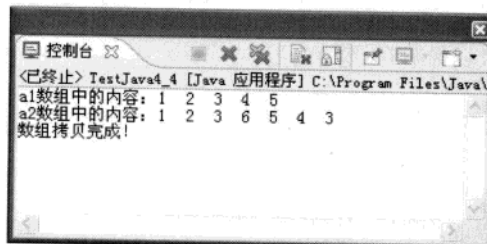
```

15         for(int i=0;i<a2.length;i++)           //输出 a2 数组中的内容
16             System.out.print(a2[i] + " ");
17         System.out.println("\n 数组拷贝完成! ");
18     }
19 }

```

【运行结果】

保存并运行程序，结果如图所示。



`System.arraycopy(source,0,dest,0,x)`语句的意思是：复制源数组从下标 0 开始的 x 个元素到目标数组，从目标数组的下标 0 所对应的位置开始存取。

【范例 8-5】 数组的排序（代码 8-5.java）。

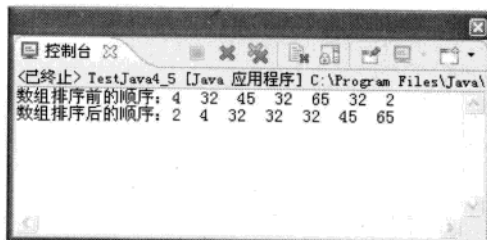
```

01 // 以下程序是数组的排序操作，在这里使用了 sort 方法对数组进行排序
02 import java.util.*;
03 public class TestJavasort
04 {
05     public static void main(String[] args)
06     {
07         int a[] = {4,32,45,32,65,32,2};
08
09         System.out.print("数组排序前的顺序: ");
10         for(int i=0;i<a.length;i++)
11             System.out.print(a[i] + " ");
12         Arrays.sort(a);           // 数组的排序方法
13         System.out.print("\n 数组排序后的顺序: ");
14         for(int i=0;i<a.length;i++)
15             System.out.print(a[i] + " ");
16     }
17 }

```

【运行结果】

保存并运行程序，结果如图所示。



程序第 12 行的 `Arrays.sort(数组名)` 为数组排序的操作，但这个方法在 `java.util` 这个包里面，所以在使用的时候需要先将它导入。至于包的概念，以后会介绍。

8.2 二维数组

▶ 本节视频教学录像：3 分钟

虽然用一维数组可以处理一般简单的数据，但是在实际应用中仍显不足，所以 Java 也提供有二维数组以及多维数组供程序设计人员使用。学会了如何使用一维数组后，再来看看二维数组的使用方法。

8.2.1 二维数组的声明与分配内存

二维数组声明的方式和一维数组类似，内存的分配也一样是用 `new` 这个关键字。其声明与分配内存的格式如下。

```
数据类型 数组名[][] ;
数组名 = new 数据类型[行的个数][列的个数] ;
```

同样，可以用较为简洁的方式来声明数组，其格式如下。

```
数据类型 数组名[][] = new 数据类型[行的个数][列的个数] ;
```

如果想直接在声明时就对数组赋初值，可以利用大括号完成。只要在数组的声明格式后面再加上所赋的初值即可，如下面的格式。

```
数据类型 数组名[][] = {
    {第 0 行初值},
    {第 1 行初值},
    ...
    {第 n 行初值},
};
```

需要特别注意的是，用户不需要定义数组的长度，因此在数组名后面的中括号里不必填入任何的内容。此外，在大括号内还有几组大括号，每组大括号内的初值会依序指定给数组的第 0、1、…、n 行元素。下面是关于数组 `num` 声明及赋初值的例子。

```
int num[][] = {
```

104

```
{23,45,21,45},           // 二维数组的初值赋值
{45,23,46,23}
};
```

语句中声明了一个整型数组 num，数组有 2 行 4 列共 8 个元素，大括号里的几组初值会分别依序指定给各行里的元素存放，num[0][0]为 23，num[0][1]为 45，…，num[1][3]为 23。

1. 每行的元素个数不同的二维数组

值得一提的是，Java 允许二维数组中每行的元素个数均不相同，这点与一般的程序语言不同。例如，下面的语句是声明整型数组 num 并赋初值，而初值的赋值指明了 num 具有三行元素，其中第 1 行有 4 个元素，第 2 行有 3 个元素，第 3 行则有 5 个元素。

```
int num[][] = {
    {42,54,34,67},
    {33,34,56},
    {12,34,56,78,90}
};
```

2. 取得二维数组的行数与特定行的元素的个数

在二维数组中，若想取得整个数组的行数，或者是某行元素的个数，则可利用“.length”来获取。其语法如下。

数组名.length	// 取得数组的行数
数组名[行的索引].length	// 取得特定行元素的个数

也就是说，如要取得二维数组的行数，只要用数组名加上“.length”即可；如要取得数组中特定行的元素的个数，则须在数组名后面加上该行的索引值，再加上“.length”，如下面的程序片段。

num.length;	// 计算数组 num 的行数，其值为 3
num[0].length	// 计算数组 num 的第 1 行元素的个数，其值为 4
num[2].length	// 计算数组 num 的第 3 行元素的个数，其值为 5

8.2.2 二维数组元素的引用及访问

二维数组元素的输入与输出方式与一维数组相同，看看下面这个范例。

【范例 8-6】 二维数组的静态赋值（代码 8-6.java）。

```
01 // 二维数组的使用说明，这里采用静态赋值的方式
02 public class TestJavaerwei
03 {
04     public static void main(String args[])
05     {
```

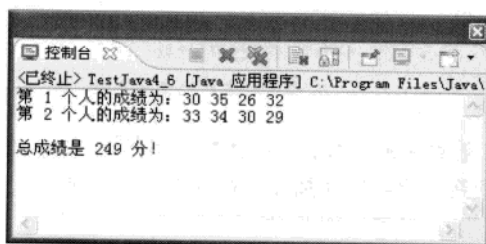
```

06      int i,j,sum=0;
07      int num[][]={{30,35,26,32},{33,34,30,29}}; // 声明数组并设置初值
08
09      for(i=0;i<num.length;i++)                // 输出销售量并计算总销售量
10      {
11          System.out.print("第 "+(i+1)+" 个人的成绩为: ");
12          for(j=0;j<num[i].length;j++)
13          {
14              System.out.print(num[i][j]+" ");
15              sum+=num[i][j];
16          }
17          System.out.println();
18      }
19      System.out.println("\n 总成绩是 "+sum+" 分!");
20  }
21  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 6 行声明整数变量 *i*、*j* 作为外层与内层循环控制变量及数组的索引，*i* 控制行的元素，*j* 控制列的元素；而 *sum* 则用来存放所有数组元素值的和，也就是总成绩。

第 7 行声明一整型数组 *num*，并对数组元素赋初值，该整型数组共有 8 个元素。

第 9~18 行输出数组里各元素的内容，并进行成绩汇总。

第 19 行输出 *sum* 的结果，即总成绩。

8.3 多维数组

本节视频教学录像：7 分钟

要想提高数组的维数，只要在声明数组的时候将索引与中括号再加一组即可，所以三维数组的声明为 `int A[][][]`，而四维数组为 `int A[][][][]`，依此类推。

下面以三维数组为例，在声明数组时即赋初值，再将其元素值输出并计算总和。

【范例 8-7】 三维数组的使用方法（代码 8-7.java）。

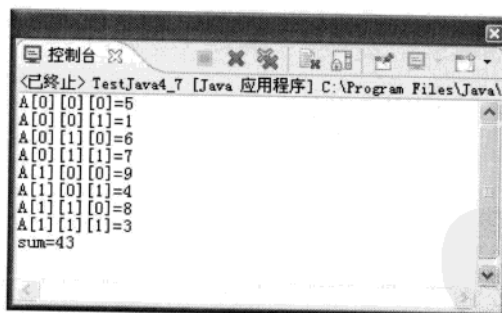
```

01 // 下面的程序说明了三维数组的使用方法，要输出数组的内容需要采用三重循环
02 public class TestJavasanwei
03 {
04     public static void main(String args[])
05     {
06         int i,j,k,sum=0;
07         int A[][][]={{5,1},{6,7}},{9,4},{8,3}}; // 声明数组并设置初值
08         // 三维数组的输出需要采用三层 for 循环方式输出
09         for(i=0;i<A.length;i++) // 输出数组内容并计算总和
10             for(j=0;j<A[i].length;j++)
11                 for(k=0;k<A[i][j].length;k++)
12                 {
13                     System.out.print("A["+i+""]["+j+""]["+k+""]="");
14                     System.out.println(A[i][j][k]);
15                     sum+=A[i][j][k];
16                 }
17         System.out.println("sum="+sum);
18     }
19 }

```

【运行结果】

保存并运行程序，结果如图所示。



由于使用的是三维数组，所以嵌套循环有 3 层。

8.4 练一练

一、填空题

1. 使用 Java 数组的两个步骤为_____和_____。
2. 在 Java 中，数组元素下标从_____开始。

3. 声明整型数组 number 的语句为_____。

二、简答题

简述 Java 中使用数组的步骤。

8.5 跟我上机

编写程序，对 `int a[] = {25, 24, 12, 76, 98, 101, 90, 28}` 数组进行排序。

第 2 篇

核心技术

在了解 Java 的基本概念之后，本篇将详细介绍 Java 编程的核心技术，包括类和对象、类库、包、异常处理、类集框架、枚举和 Annotation 等。通过本篇的学习，读者将对使用 Java 编程有更高的掌握水平。

- ▶ 第 9 章 面向对象设计——类和对象
- ▶ 第 10 章 类的封装、继承与多态
- ▶ 第 11 章 抽象类与接口
- ▶ 第 12 章 关于类的专题研究
- ▶ 第 13 章 储存类的仓库——Java 常用类库
- ▶ 第 14 章 包及访问权限
- ▶ 第 15 章 异常处理
- ▶ 第 16 章 Java 类集框架
- ▶ 第 17 章 JDK 1.5 以上版本的新功能——枚举
- ▶ 第 18 章 给编译器看的注释——Annotation

第 9 章

面向对象设计——类和对象



本章视频教学录像：2 小时 13 分钟

类是Java中的一个重要概念。和对象一样，要想熟练使用Java语言，就一定要掌握类的使用。本章介绍Java语言中的对象和类，其封装性、继承性和多态性的概念，声明创建类和对象的方法以及构造方法。

本章要点（已掌握的在方框中打勾）

- ☐ 了解类和对象的相关概念
- ☐ 掌握声明以及创建类和对象的方法
- ☐ 掌握对象的比较方法
- ☐ 掌握类的方法的使用及调用



到目前为止,前面介绍的Java语法都属于Java语言的最基本功能,其中包括数据类型和程序控制语句、循环语句等。但随着计算机的发展,面向对象的概念也随之孕育而生。类(class)是面向对象程序设计最重要的概念之一。要深入了解Java程序语言,一定要树立面向对象程序设计的观念。从本章开始学习Java程序中类的设计。

9.1 面向对象程序设计的基本概念

本节视频教学录像: 10 分钟

面向对象其实是现实世界模型的自然延伸。可以将现实世界中的任何实体都看做是对象,对象之间通过消息相互作用。另外,现实世界中的任何实体都可归属于某类事物,任何对象都是某一类事物的实例。如果说传统的过程式编程语言是以过程为中心、以算法为驱动的话,面向对象的编程语言则是以对象为中心、以消息为驱动。用公式表示,过程式编程语言为:程序=算法+数据,面向对象编程语言为:程序=对象+消息。

所有的面向对象编程语言都支持3个概念,即封装、多态性和继承。现实世界中的对象均有属性和行为,映射到计算机程序上,属性则表示对象的数据,行为表示对象的方法(其作用是处理数据或同外界交互)。所谓封装,就是用一个自主式框架把对象的数据和方法连在一起形成一个整体。可以说,对象是支持封装的手段,是封装的基本单位。类描述了一组有相同特性(属性)和相同行为(方法)的对象。在程序中,类实际上就是数据类型,例如整数、小数等。整数也有一组特性和行为。面向过程的语言与面向对象的语言的区别就在于,面向过程的语言不允许程序员自己定义数据类型,而只能使用程序中内置的数据类型。而为了模拟真实世界,为了更好地解决问题,我们往往需要创建解决问题所必需的数据类型。

9.1.1 对象

对象的特征分为静态特征和动态特征两种。静态特征指对象的外观、性质、属性等,动态特征指对象具有的功能、行为等。客观事物是错综复杂的,但人们总是从某一目的出发,运用抽象分析的能力,从众多的特征中抽取最具代表性、最能反映对象本质的若干特征加以详细研究。

人们将对象的静态特征抽象为属性,用数据来描述,在Java语言中称之为变量;人们将对象的动态特征抽象为行为,用一组代码来表示,完成对数据的操作,在Java语言中称之为方法。一个对象由一组属性和一组对属性进行操作的方法构成。

9.1.2 类

将具有相同属性及相同行为的一组对象称为类。广义地讲,具有共同性质的事物的集合就称为类。

在面向对象程序设计中,类是一个独立的单位,它有一个类名,其内部包括成员变量,用于描述对象的属性;还包括类的成员方法,用于描述对象的行为。在Java程序设计中,类被认为是一种抽象的数据类型,这种数据类型不但包括数据,还包括方法,这大大地扩充了数据类型的概念。

类是一个抽象的概念,要利用类的方式来解决实际问题,必须用类创建一个实例化的类对象,然

后通过类对象去访问类的成员变量，去调用类的成员方法来实现程序的功能。就如同“汽车”本身是一个抽象的概念，只有使用了一辆具体的汽车，才能感受到汽车的功能。

一个类可创建多个类对象，它们具有相同的属性模式，但可以具有不同的属性值。Java 程序为每一个类对象都开辟了内存空间，以便保存各自的属性值。

面向对象的程序设计有以下 3 个主要特征。

- 封装性
- 继承性
- 多态性

9.1.3 封装性

封装性将尽可能对外界公布一个有限的界面，而将其细节隐藏起来，与其他对象的交往通过这个界面进行。这就是我们常说的信息隐藏性。

一个公司（或企业）的经营活动可以理解为一个“封装性”的概念。该公司的基本活动可以由其各部门（基本数据）和调度部门（对此数据操作的过程）或共同完成的产品（函数）结合起来。各部门之间的（外界）联系是由公司总调度（或领导）和各部门调度（或部门领导）实现的。

对“封装”性概念可从以下 3 个方面理解。

(1) 所有软件的内部都应有明确的范围、清楚的外部边界，这就像公司的各部门有明确的职责范围、各部门之间有确定的权限界限一样。

(2) 每个软件部件都应具有友好的界面或接口，以表明软件部件之间的相互作用和相互联系，这就像公司的各部门相互间的衔接工序及各部门之间有联系与制约一样。

(3) 完成、保护和隐藏软件的内部实现，用户不必了解其具体的实现，这就像一个工厂的各个车间生产不同的零部件，本车间完成自己的工作，对外提供成品零部件，使用零部件者不必知道其生产过程一样。

有了封装性，软件设计人员就可以集中精力考虑开发系统各模块之间的关系等重大问题，而模块内部的实现则可由程序设计人员研究与完善，这样可以充分保证模块的质量和可靠性，也支持软件工程化思想。

9.1.4 继承性

如果类 B 具有类 A 的全部属性和方法，而且又具有自己特有的某些属性和方法，则把类 A 称做一般类，而把类 B 称做特殊类。

在面向对象程序设计中运用继承原则，就是在每个由一般类和特殊类形成的一般 — 特殊结构中，把一般类的对象实例和所有特殊类的对象实例都共同具有的属性和操作一次性地在一般类中进行显式的定义，在特殊类中不再重复地定义一般类中已经定义的东西，但是在语义上，特殊类却会自动地、隐含地拥有它的一般类（以及所有更上层的一般类）中定义的属性和操作。

特殊类的对象拥有其一般类的全部或部分属性与方法，称做特殊类对一般类的继承。

继承所表达的就是一种对象之间的相交关系，它使得某类对象可以继承另外一类对象的数据成员和成员方法。

若类 B 继承类 A，则属于 B 的对象便具有类 A 的全部或部分性质（数据属性）和功能（操作）。

我们称被继承的类 A 为基类、父类或超类，而称继承类 B 为 A 的派生类或子类。

继承避免了对一般类和特殊类之间共同特征进行的重复描述。

继承具有以下特征。

(1) 继承关系是传递的。继承是在一些比较一般的类的基础上构造、建立和扩充新类的最有效的手段。

(2) 继承简化了人们对事物的认识和描述，能清晰体现相关类间的层次结构关系。

(3) 提供软件复用功能。

(4) 通过增强一致性来减少模块间的接口和界面，大大增加程序的易维护性。

(5) 提供多重继承机制。从理论上说，一个类可以是多个一般类的特殊类，它可以从多个一般类中继承属性和方法，这便是多重继承。而 Java 出于安全性和可靠性的考虑，仅支持单重继承，而通过使用接口机制来实现多重继承。

9.1.5 多态性

多态是面向对象程序设计的又一个重要特征。多态是允许程序中出现重名现象。Java 语言中含有方法重载与成员覆写两种形式的多态。

方法重载：在一个类中，允许多个方法使用同一个名字，但方法的参数不同，完成的功能也不同。

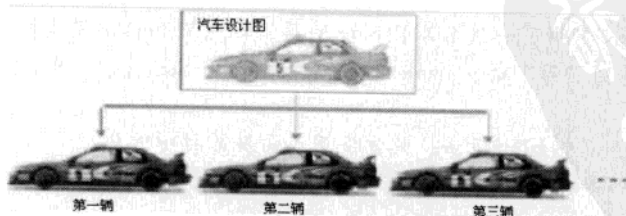
成员覆写：子类与父类允许具有相同的变量名称，数据类型不同，允许具有相同的方法名称，但完成的功能不同。

多态的特性使程序的抽象程度和简捷程度更高，有助于程序设计人员对程序的分组协同开发。

9.2 类

本节视频教学录像：50 分钟

面向对象的编程思想力图使在计算机语言中对事物的描述与现实世界中该事物的本来面目尽可能地一致，类（class）和对象（object）就是面向对象方法的核心概念。类是对某一类事物的描述，是抽象的、概念上的定义；对象是实际存在的该类事物的个体，因而也称做实例（Instance）。下图所示是一个说明类与对象的典型范例。



图中汽车设计图就是“类”，由这个图纸设计出来的若干的汽车就是按照该类产生的“对象”。可见，类描述了对对象的属性和对象的行为，类是对象的模板。对象是类的实例，是一个实实在在的个体，一个类可以对应多个对象。可见，如果将对象比做汽车，那么类就是汽车的设计图纸，所以面向对象程序设计的重点是类的设计，而不是对象的设计。

同一个类按同种方法产生出来的多个对象，其开始的状态都是一样的，但是修改其中一个对象的时候，其他的对象是不会受到影响的，比如修改第 1 辆汽车的时候，其他的汽车是不会受到影响的。

9.2.1 类的声明

在使用类之前，必须先声明它，然后才可利用所声明的类来声明变量，并创建对象。类声明的语法如下。

```
class 类名称
{
    //类的成员变量
    //类的方法
}
```

可以看到，声明类使用的是 class 关键字。声明一个类时，在 class 关键字后面加上类的名称，这样就创建了一个类，然后在类的里面定义成员变量和方法。

下面举一个 Person 类的例子，以使读者清楚地认识类的组成。

【范例 9-1】 类的组成使用（代码 9-1.java）。

```
01 class Person
02 {
03     String name ;
04     int age ;
05     void talk()
06     {
07         System.out.println("我是: "+name+", 今年: "+age+"岁");
08     }
09 }
```

【代码详解】

程序首先用 class 声明了一个名为 Person 的类，在这里 Person 是类的名称。

第 3、4 行先声明了两个属性 name 和 age，name 为 String（字符串类型）型，age 为 int（整型）型。

第 5~8 行声明了一个 talk() 方法，此方法用于向屏幕打印信息。

为了更好地说明类的关系，请看下图。

Person
+ name : String
+ age : int
+ talk () : void



提示：读者可以发现本例中，声明类 `Person` 时，类名中单词的首字母是大写，这是规定的一种符合标准的写法，在本书以后的范例中都将采用这种写法。

9.2.2 类的定义

在声明一个类后，还需要对类进行定义，定义一个类后就定义了一个功能模块。定义类的语法如下。

```
class 类名称
{
    数据类型 属性 ;

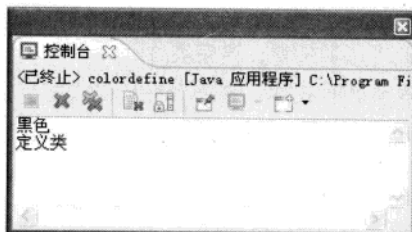
    返回值的数据类型 方法名称 ( 参数 1, 参数 2, ... )
    {
        程序语句 ;
        return 表达式 ;
    }
}
```

定义一个类后，就可以创建类的实例了，创建类实例通过 `new` 关键字完成。下面通过一个实例讲解如何定义并使用类。

【范例 9-2】 类的定义使用（代码 9-2.java）。

```
01 class colordefine
02 {
03     String color = "黑色";
04     void getMes()
05     {
06         System.out.println("定义类");
07     }
08     public static void main(String args[]) {
09         {
10             colordefine b = new colordefine();
11             System.out.println(b.color);
12             b.getMes();
13         }
14     }
15 }
```

程序运行结果如图所示。



9.3 对象

▶ 本节视频教学录像：41 分钟

在上面的范例中，已经创建好了一个 `Person` 的类，相信类的基本形式读者应该已经很清楚了。但是在实际中单单有类是不够的，类提供的只是一个模板，必须依照它创建出对象之后才可以使用。

9.3.1 对象的声明

下面定义了由类产生对象的基本形式。

```
类名 对象名 = new 类名();
```

了解上述的概念之后，便可动手编写程序了。创建属于某类的对象，需要通过下面两个步骤实现。

- (1) 声明指向“由类所创建的对象”的变量。
- (2) 利用 `new` 创建新的对象，并指派给先前所创建的变量。

举例来说，如果要创建 `Person` 类的对象，可用下列语句实现。

```
Person p ;           // 先声明一个 Person 类的对象 p
p = new Person();     // 用 new 关键字实例化 Person 的对象 p
```

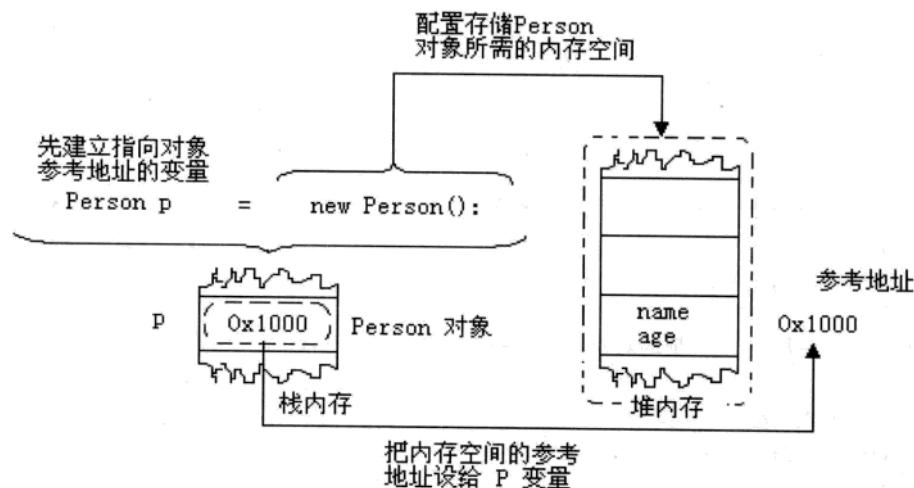
当然也可以用下面的这种形式来声明变量。

```
Person p = new Person(); // 声明 Person 对象 p 并直接实例化此对象
```



提示：对象只有在实例化之后才能被使用，而实例化对象的关键字就是 `new`。

对象实例化的过程如图所示。



从图中可以看出，当语句执行到 Person p 的时候，只是在栈内存中声明了一个 Person 的对象 p，但是这个时候 p 并没有在堆内存中开辟空间，所以这个时候的 p 是一个未实例化的对象。用 new 关键字实际上就是开辟堆内存，把堆内存的引用赋予 p，这个时候的 p 才称为一个实例化对象。

9.3.2 对象的使用

如果要访问对象里的某个成员变量或方法，可以通过下面的语法来实现。

访问属性：对象名称.属性名

访问方法：对象名称.方法名()

例如想访问 Person 类中的 name 和 age 属性，可以用如下方法来访问。

```
p.name;           // 访问 Person 类中的 name 属性
p.age;            // 访问 Person 类中的 age 属性
```

因此若想将 Person 类的对象 p 中的属性 name 赋值为“张三”，年龄赋值为 25，则可采用下面的写法。

```
p.name = "张三";
p.age = 25;
```

如果想调用 Person 中的 talk() 方法，可以采用下面的写法。

```
p.talk();          // 调用 Person 类中的 talk() 方法
```

下面是完整的程序。

```
01 class Person
```

```

02 {
03     String name ;
04     int age ;
05     void talk()
06     {
07         System.out.println("我是: "+name+", 今年: "+age+"岁");
08     }
09 }

```

【范例 9-3】 使用 Person 类的对象调用类中的属性与方法的过程（代码 9-3.java）。

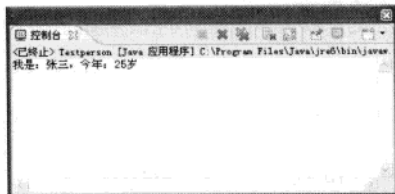
```

01 // 下面这个范例说明了使用 Person 类的对象调用类中的属性与方法的过程
02 class TestPersonDemo
03 {
04     public static void main(String[] args)
05     {
06         Person p = new Person();
07         p.name = "张三";
08         p.age = 25;
09         p.talk();
10     }
11 }

```

【运行结果】

保存并运行程序，结果如图所示。



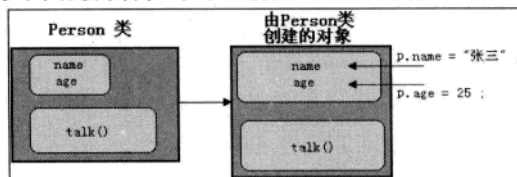
【代码详解】

第 6 行声明了一个 Person 类的实例对象 p，并直接实例化此对象。

第 7、8 行对 p 对象中的属性赋值。

第 9 行调用 talk() 方法，在屏幕上输出信息。

对照上述程序代码与下图的内容，即可了解到 Java 是如何对对象成员进行访问操作的。



9.3.3 对象的比较

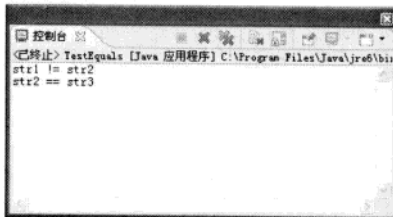
有两种方式可用于对象间的比较，即“==”运算符与 equals()方法。“==”运算符用于比较两个对象的内存地址值是否相等，equals()方法用于比较两个对象的内容是否一致。

【范例 9-4】 “==”运算符用于比较（代码 9-4.java）。

```
01 public class TestEquals
02 {
03     public static void main(String[] args)
04     {
05         String str1 = new String("java");
06         String str2 = new String("java");
07         String str3 = str2;
08         if(str1==str2)
09         {
10             System.out.println("str1 == str2");
11         }
12         else
13         {
14             System.out.println("str1 != str2");
15         }
16         if(str2==str3)
17         {
18             System.out.println("str2 == str3");
19         }
20         else
21         {
22             System.out.println("str2 != str3");
23         }
24     }
25 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

由程序的输出结果可以发现，str1 不等于 str2，有些读者可能会问，str1 与 str2 的内容完全一样，为什么会不等于呢？读者可以发现程序的第 5 行和第 6 行分别实例化了 String 类的两个对象，此时，这两个对象指向不同的内存空间，所以它们的内存地址是不一样的。这个时候程序中是用的“==”比较，比较的是内存地址值，所以输出 str1!=str2。程序第 7 行将 str2 的引用赋给 str3，这个时候就相当于 str3 也指向了 str2 的引用，此时这两个对象指向的是同一内存地址，所以比较值的结果是 str2==str3。

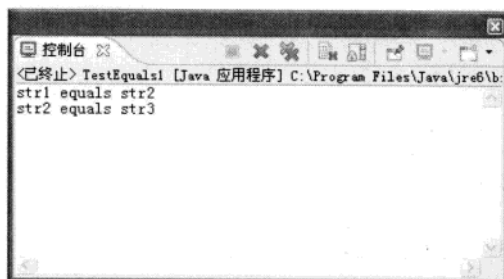
读者可能会问，那该如何去比较里面的内容呢？这就需要采用另外一种方式——“equals”。请看下面的程序，程序中用 TestEquals1.java 修改自程序 TestEquals.java。

【范例 9-5】 equals 方法用于对象比较（代码 9-5.java）。

```
01 public class TestEquals1
02 {
03     public static void main(String[] args)
04     {
05         String str1 = new String("java");
06         String str2 = new String("java");
07         String str3 = str2;
08         if(str1.equals(str2))
09         {
10             System.out.println("str1 equals str2");
11         }
12         else
13         {
14             System.out.println("str1 not equals str2");
15         }
16         if(str2.equals(str3))
17         {
18             System.out.println("str2 equals str3");
19         }
20         else
21         {
22             System.out.println("str2 note equals str3");
23         }
24     }
25 }
```

【运行结果】

保存并运行程序，结果如图所示。



这个时候可以发现，在程序中将比较的方式换成了 `equals`，而且调用 `equals()` 方法的是 `String` 类的对象，所以可以知道 `equals` 是 `String` 类中的方法。在这里读者一定要记住：“`==`”是比较内存地址值的，而“`equals`”是比较内容的。

9.3.4 对象数组的使用

在前面已介绍过如何以数组来保存基本数据类型的变量。相同的，对象也可以用数组来存放，可通过下面两个步骤来实现。

(1) 声明类类型的数组变量，并用 `new` 分配内存空间给数组。

(2) 用 `new` 产生新的对象，并分配内存空间给它。

例如要创建 3 个 `Person` 类类型的数组元素，语法如下。

```
Person p[];           // 声明 Person 类类型的数组变量
p = new Person[3];     // 用 new 分配内存空间
```

创建好数组元素之后，便可把数组元素指向由 `Person` 类所产生的对象。

```
p[0] = new Person ();
p[1] = new Person ();
p[2] = new Person ();
```

此时，`p[0]`、`p[1]`、`p[2]`是属于 `Person` 类类型的变量，它们分别指向新建对象的内存参考地址。当然也可以写成如下形式。

```
Person p[] = new Person[3]; // 创建对象数组元素，并分配内存空间
```

当然，也可以利用 `for` 循环来完成对象数组内的初始化操作，此方式属于动态初始化。

```
for(int i=0;i<p.length;i++)
{
    p[i] = new Person();
}
```

或者采用静态方式初始化对象数组。

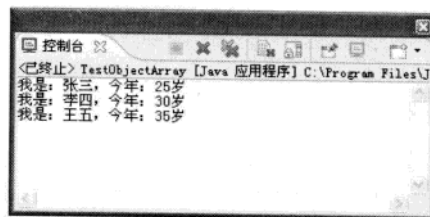
```
Person p[] = {new Person(),new Person(),new Person()};
```

【范例 9-6】 用静态方式初始化对象数组（代码 9-6.java）。

```
01 class Person
02 {
03     String name ;
04     int age ;
05     public Person()
06     {
07     }
08     public Person(String name,int age)
09     {
10         this.name = name ;
11         this.age = age ;
12     }
13     public String talk()
14     {
15         return "我是: "+this.name+", 今年: "+this.age+"岁" ;
16     }
17 }
18 public class TestObjectArray
19 {
20     public static void main(String[] args)
21     {
22         Person p[] = {
23             new Person("张三",25),new Person("李四",30),new Person("王五",35)
24         };
25         for(int i=0;i<p.length;i++)
26         {
27             System.out.println(p[i].talk());
28         }
29     }
30 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

程序第 22~24 行用静态声明方式声明了 3 个对象的 Person 类的对象数组。

程序第 25~28 行用 for 循环输出所有对象，并分别调用 talk() 方法打印信息。

9.4 类的属性

类的基本组成部分包含属性和方法。先来讲解类的属性，类的属性也称为字段或成员变量，不过习惯上将它称为属性。

9.4.1 属性的定义

属性的定义语法如下。

```
[public|protected|private|default] [(static [final])|(final [static])] className prorotypeName
```

但是要注意，下面的定义方法有些是错误的。

```
public class A{public class B{public static String _str;}} //错误，非静态内部类不能定义静态属性
public class A{public class B{public static final String _str;}} //正确，非静态内部类可以定义静态常量属性
public class A{public class B{public static void method(){} }} //错误，非静态内部类不能定义静态方法
public class A{public static class B{public static void method(){} }} //正确，静态内部类可以定义静态方法
public class A{public static class B{public void method(){} }} //正确，静态内部类可以定义非静态方法
public class A{public static class B{public static String _str;}} //正确，静态内部类可以定义静态属性
public class A{public class B{public static class C{}} } //错误，非静态内部类不能定义静态内部类
public class A{public static class B{public class C{}} } //正确，静态内部类可以定义非静态内部类
public class A{public static class B{public static class C{}} } //正确，静态内部类可以定义静态内部类
```

类的属性定义规则如下。

- (1) 类的属性是变量。
- (2) 类的属性的类型可以是基本类型，也可以是引用类型。
- (3) 类的属性的命名规则，首单词的首字母小写，其余单词的首字母大写。

类变量和成员变量的区别在于：类变量就是 static 修饰的变量，它们被类的实例所共享，就是说一个实例改变了这个值，其他的实例也会受到影响；成员变量则是实例所私有的，只有实例本身可以改变它的值。

9.4.2 属性的使用

下面通过一个实例来讲解类的属性的使用，通过这个实例可以看出在 Java 中类属性和对象属性的不同使用方法。

【范例 9-7】 类的属性组使用（代码 9-7.java）。

```
01 public class Test {
02
03
04     static String a = "string-a";
05     static String b;
06
07
08     String c = "string-c";
09     String d;
10
11     static {
12         printStatic("before static");
13         b = "string-b";
14         printStatic("after static");
15     }
16
17     public static void printStatic(String title) {
18         System.out.println("-----" + title + "-----");
19         System.out.println("a = \"\" + a + "\"");
20         System.out.println("b = \"\" + b + "\"");
21     }
22
23     public Test() {
24         print("before constructor");
25         d = "string-d";
26         print("after constructor");
27     }
28
29     public void print(String title) {
30         System.out.println("-----" + title + "-----");
31         System.out.println("a = \"\" + a + "\"");
32         System.out.println("b = \"\" + b + "\"");
33         System.out.println("c = \"\" + c + "\"");
34         System.out.println("d = \"\" + d + "\"");
35     }
36
37     public static void main(String[] args) {
38         new Test();
39     }
```

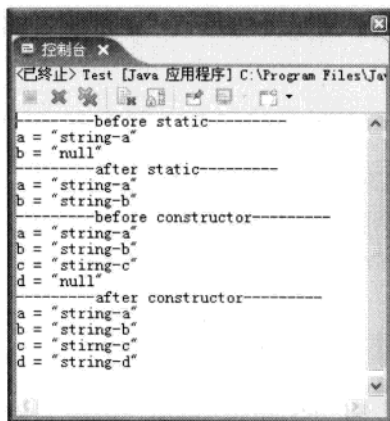
```

40
41 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

由此可以看出 Java 类属性和对象属性的初始化顺序如下。

- (1) 类属性（静态变量）定义时的初始化，如范例中的 `static String a = "string-a"`。
- (2) `static` 块中的初始化代码，如范例中的 `static {}` 中的 `b = "string-b"`。
- (3) 对象属性（非静态变量）定义时的初始化，如范例中的 `String c = "string-c"`。
- (4) 构造方法（函数）中的初始化代码，如范例构造方法中的 `d = "string-d"`。

9.5 类的方法

▶ 本节视频教学录像：32 分钟

类的方法是类的任一个特定实例都能调用的方法，作用范围是整个类，而不是类的某个特定实例。类方法也称做静态方法，在类中定义的方法叫做该类的成员。

9.5.1 方法的定义

所有的方法均在类中声明。定义方法的一般形式如下。

```

type name(参数列表){
    /方法主体
}

```

`type` 指定了由方法返回的数据类型。它可以是任意有效的类型，包括创建的类类型。如果方法没有返回值，则其返回类型必须是 `void`。方法的名字由 `name` 指定，这个名字可以是除了那些

在当前作用域中已经使用的标识符之外的任意合法标识符。参数列表是由类型、标识符对组成的序列，每对之间用逗号分开。参数实际上是方法被调用时接收传递过来的参数值的变量。如果方法没有参数，那么参数表就是空的。

9.5.2 方法的使用

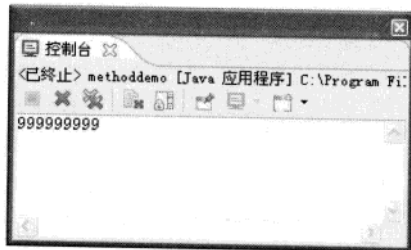
下面通过一个实例讲解方法的使用。在实例中首先创建一个方法，方法完成乘法计算功能，然后在主函数中调用该方法。

【范例 9-8】 方法的使用（代码 9-8.java）。

```
01 public class methoddemo {  
02     int a = 12345679 , b =81;  
03     public void times(int i, int j)  
04     {  
05         System.out.println(i*j);  
06     }  
07     public static void main(String args[])  
08     {  
09         methoddemo m = new methoddemo();  
10         m.times(m.a, m.b);  
11     }  
12 }
```

【运行结果】

保存并运行程序，结果如图所示。



9.5.3 构造方法

在 Java 程序里，构造方法所完成的主要工作是帮助新创建的对象赋初值。可将构造方法视为一种特殊的方法，它的定义方式与普通方法类似，其语法如下。

```
class 类名称  
{  
    访问权限 类名称 (类型 1 参数 1, 类型 2 参数 2, ... )
```

```

{
    程序语句 ;
    ...    // 构造方法没有返回值
}
}

```

在使用构造方法的时候需注意以下两点。

- (1) 它具有与类名相同的名称。
- (2) 它没有返回值。

构造方法除了没有返回值，且名称必须与类的名称相同之外，它的调用时机也与一般的方法不同。一般的方法是在需要时才调用，而构造方法则是在创建对象时自动调用，并执行构造方法的内容。因此，构造方法无需在程序中直接调用，而是在对象产生时自动执行。

基于上述构造方法的特性，可利用它来对对象的数据成员做初始化的赋值。所谓初始化就是为对象赋初值。

下面的程序 TestConstruct.java 说明了构造方法的使用。

【范例 9-9】 Java 中构造方法的使用（代码 9-9.java）。

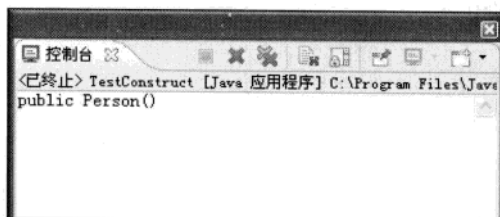
```

01  class Person
02  {
03      public Person()           // Person 类的构造方法
04      {
05          System.out.println("public Person()");
06      }
07  }
08
09  public class TestConstruct
10  {
11      public static void main(String[] args)
12      {
13          Person p = new Person();
14      }
15  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~7 行声明了一个 Person 类，此类中只有一个 Person 的构造方法。

第 3~6 行声明了一个 Person 类的构造方法，此方法只含有一个输出语句。

第 13 行实例化了一个 Person 类的对象 p，此时会自动调用 Person 中的无参构造方法，即在屏幕上打印信息。

【范例分析】

从此程序中读者不难发现，在类中声明的构造方法，会在实例化对象时自动调用。

读者可能会问，在之前的程序中用同样的方法来产生对象，但是在类中并没有声明任何构造方法，而程序不也一样可以正常运行吗？

实际上，读者在执行 javac 编译 java 程序的时候，如果在程序中没有明确声明一个构造方法的话，系统会自动为类加入一个无参的且什么都不做的构造方法，类似于下面代码。

```
public Person()  
{  
}
```

所以，之前所使用的程序虽然没有明确地声明构造方法，也是可以正常运行的。

9.5.4 构造方法的重载

在 Java 里，不仅普通方法可以重载，构造方法也可以重载。只要构造方法的参数个数不同，或是类型不同，便可定义多个名称相同的构造方法。这种做法在 Java 中是常见的，请看下面的程序。

【范例 9-10】 构造方法的重载（代码 9-10.java）。

```
01 class Person  
02 {  
03     private String name ;  
04     private int age ;  
05     public Person(String n,int a)  
06     {  
07         name = n ;  
08         age = a ;  
09         System.out.println("public Person(String n,int a)");  
10     }  
11     public String talk()  
12     {  
13         return "我是: "+name+"，今年: "+age+"岁";  
14     }  
15 }  
16
```

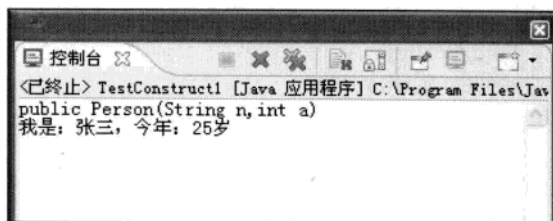
```

17 public class TestConstruct1
18 {
19     public static void main(String[] args)
20     {
21         Person p = new Person("张三",25);
22         System.out.println(p.talk());
23     }
24 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第1~15行声明了一个名为 Person 的类，里面有 name 与 age 两个私有属性，和一个 talk() 方法。

第5~10行，在 Person 类中声明了一个含有两个参数的构造方法，此构造方法用于将传入的值赋给 Person 类的属性。

第21行调用 Person 类中含有两个参数的构造方法（new Person("张三",25)），同时将姓名和年龄传到类里，分别对各个属性赋值。

第22行调用 Person 类中的 talk() 方法打印信息。

【范例分析】

从本程序可以发现，构造方法的基本作用就是对类中的属性初始化，在程序产生类的实例对象时，将需要的参数由构造方法传入，之后再由构造方法为其内部的属性进行初始化。这是在一般开发中经常使用的技巧，但是这里有一个问题是读者应该注意的，请看下面的程序。

【范例 9-11】 构造方法的使用范例 1（代码 9-11.java）。

```

01 class Person
02 {
03     private String name ;
04     private int age ;
05     public Person(String n,int a)
06     {
07         name = n ;
08         age = a ;

```

```

09      System.out.println("public Person(String n,int a)");
10  }
11  public String talk()
12  {
13      return "我是: "+name+", 今年: "+age+"岁";
14  }
15  }
16
17  public class TestConstruct2
18  {
19      public static void main(String[] args)
20      {
21          Person p = new Person();
22          System.out.println(p.talk());
23      }
24  }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

可以发现，在编译程序第 21 行（`Person p = new Person()`）时发生了错误，这个错误说找不到 `Person` 类的无参构造方法。在前面曾经提过，如果程序中没有声明构造方法，程序就会自动声明一个无参的且什么都不做的构造方法，可是现在却发生了找不到无参构造方法的问题，这是为什么？读者可以发现第 5~10 行已经声明了一个含有两个参数的构造方法。在 Java 程序中只要明确地声明了构造方法，那么默认的构造方法就不会被自动生成。而要解决这一问题，只需要简单地修改一下 `Person` 类就可以了，可以在 `Person` 类中明确地声明一无参的且什么都不做的构造方法。

【范例 9-12】 构造方法的使用范例 2（代码 9-12.java）。

```

01  class Person
02  {
03      private String name ;

```

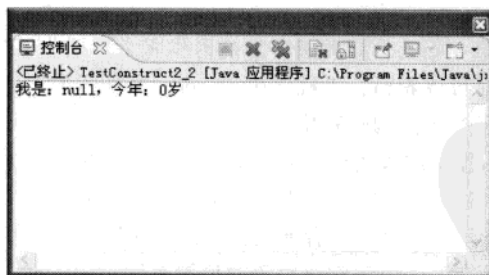
```

04     private int age ;
05     public Person()
06     {}
07     public Person(String n,int a)
08     {
09         name = n ;
10         age = a ;
11         System.out.println("public Person(String n,int a)" );
12     }
13     public String talk()
14     {
15         return "我是: "+name+", 今年: "+age+"岁";
16     }
17 }
18
19 public class TestConstruct2
20 {
21     public static void main(String[] args)
22     {
23         Person p = new Person() ;
24         System.out.println(p.talk());
25     }
26 }

```

【运行结果】

保存并运行程序，结果如图所示。



可以看见，在程序的第 5、6 行声明了一无参的且什么都不做的构造方法，此时再编译程序的话，就可以正常编译而不会出现错误了。

9.5.5 构造方法的私有

方法依实际需要，可分为 public 与 private。同样，构造方法也有 public 与 private 之分。到目前为止，所使用的构造方法均属于 public，它可以在程序的任何地方被调用，所以新创建的对

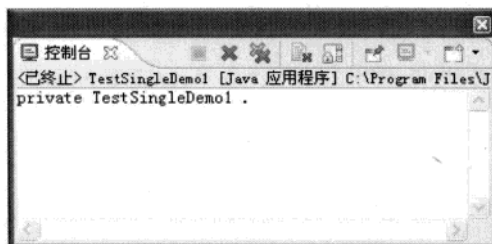
象也都可以自动调用它。如果构造方法被设为 `private`，则无法在该构造方法所在的类以外的地方被调用。

【范例 9-13】 构造方法的私有使用范例 1（代码 9-13.java）。

```
01 public class TestSingleDemo1
02 {
03     private TestSingleDemo1()
04     {
05         System.out.println("private TestSingleDemo1 .");
06     }
07     public static void main(String[] args)
08     {
09         new TestSingleDemo1();
10     }
11 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

从此程序可以看出，第 3 行在声明构造方法的时候将之声明为 `private` 类型，则此构造方法只能在本类内被调用。同时可以看出，本程序中的 `main` 方法也放在了 `TestSingleDemo1` 类的内部，所以在本类中可以自己产生实例化对象。

看过上面的程序，这么做似乎没有什么意义，因为一个类最终都会由外部去调用，如果这么做的话，岂不是所有构造方法被私有化了的类都需要这么去调用了吗？那程序岂不是会有很多的 `main()` 方法了吗？举这个例子主要是让读者清楚：构造方法虽然被私有了，但并不一定是说此类不能产生实例化对象，只是产生这个实例化对象的位置有所变化，即只能在本类中产生实例化对象。请看下面的范例。

【范例 9-14】 构造方法的私有使用范例 2（代码 9-14.java）。

```
01 class Person
02 {
03     String name ;
```

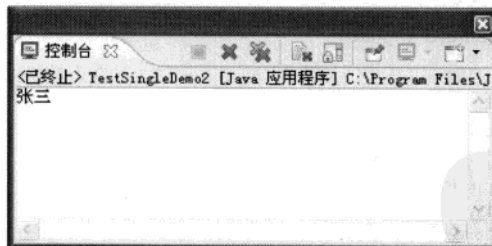
```

04      // 在本类声明一 Person 对象 p，注意此对象用 final 标记，表示不能再重新实例化
05      private static final Person p = new Person();
06      private Person()
07      {
08          name = "张三";
09      }
10      public static Person getP()
11      {
12          return p;
13      }
14  }
15
16  public class TestSingleDemo2
17  {
18      public static void main(String[] args)
19      {
20          // 声明一个 Person 类的对象
21          Person p = null;
22          p = Person.getP();
23          System.out.println(p.name);
24      }
25  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 6~9 行将 Person 类的构造方法封装起来，外部无法通过其构造方法产生实例化对象。

第 5 行声明一个 Person 类的实例化对象，此对象是在 Person 类内部实例化，所以可以调用私有构造方法。另外，此对象被标识为 static 类型，表示为一静态属性，同时此对象被私有化，另外在声明 Person 对象的时候加上了一个 final 关键字，此关键字表示 Person 的对象 p 不能被重新实例化（关于 final，请参阅第 11 章中的 11.3 一节）。

第 21 行声明一个 Person 类的对象 p，但未实例化。

第 22 行调用 Person 类中的 getP() 方法，此方法返回 Person 类的实例化对象。

【范例分析】

从程序中可以看出，无论在 Person 类的外部声明多少个对象，最终得到的都是同一个引用，因为此类只能产生一个实例对象，这种做法在设计模式中被称为单态模式。而所谓设计模式也就是在大量的实践中总结和理论化之后优选的代码结构、编程风格以及解决问题的思考方式。有兴趣的读者可以研究一下。

9.5.6 在类内部调用方法

通过上面的几个范例，读者应该清楚，在一个 Java 程序中是可以通过对象去调用类中的方法的，当然类的内部也能互相调用各自的方法，比如在下面的程序中，修改了以前的程序代码，新增加了一个公有的 say() 方法，并用这个方法去调用私有的 talk() 方法。

【范例 9-15】 在类的内部调用方法（代码 9-15.java）。

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      private void talk()
06      {
07          System.out.println("我是: "+name+"，今年: "+age+"岁");
08      }
09      public void say()
10      {
11          talk();
12      }
13      public void setName(String str)
14      {
15          name = str ;
16      }
17      public void setAge(int a)
18      {
19          if(a>0)
20              age = a ;
21      }
22      public String getName()
23      {
24          return name ;
25      }
26      public int getAge()
27      {
```

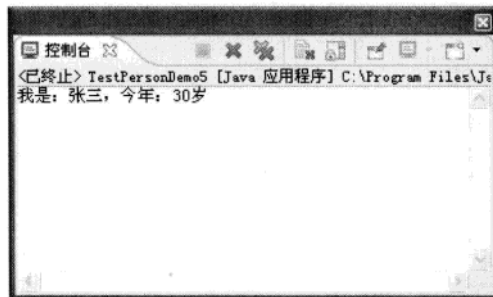
```

28         return age ;
29     }
30 }
31
32 public class TestPersonDemo5
33 {
34     public static void main(String[] args)
35     {
36         // 声明并实例化一个 Person 对象 p
37         Person p = new Person() ;
38         // 给 p 中的属性赋值
39         p.setName("张三") ;
40         // 在这里将 p 对象中的年龄属性赋值为 30 岁
41         p.setAge(30) ;
42         // 调用 Person 类中的 say()方法
43         p.say() ;
44     }
45 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 9~12 行声明一个公有方法 say()，此方法用于调用类内部的私有方法 talk()。

第 43 行调用 Person 类中的 say()方法，其实也就是调用了 Person 类中的 talk()方法。



注 意：这个时候是用 say()方法调用 talk()方法。如果非要强调对象本身的话，也可以写成“this.talk()；”的形式。

读者也许会觉得这样写有些多余，当然 this 的使用方法很多，在以后会完整地介绍。读者可自行修改上面的程序试验一下，看看结果是不是与原来的相同。

9.6 练一练

一、填空题

1. 面向对象的程序设计有 3 个主要特征：_____、_____和_____。
2. 在继承过程中，被继承的类称为_____，经继承产生的类称为_____。
3. 在 Java 中，多态的两种形式为_____和_____。

二、简答题

简述 Java 中两种形式的多态。

9.7 跟我上机

定义一个包含 name、age 和 like 属性的 Person 类，实例化并给对象赋值，然后输出对象属性。

第 10 章

类的封装、继承与多态



本章视频教学录像：1 小时 38 分钟

类的封装相当于一个黑匣子，放在黑匣子中的东西你什么也看不到。继承是类的一个重要属性，可以从一个简单的类继承出相对复杂高级的类，这样可使程序编写的工作量大大减轻。多态则可动态地对对象进行调用，使对象之间变得相对独立。本章讲解类的3大特性：封装、继承和多态。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握封装的基本概念和应用
- ☐ 掌握继承的基本概念和应用
- ☐ 掌握多态的基本概念和应用
- ☐ 掌握 super 关键字的使用
- ☐ 熟悉对象的多态性



10.1 类的封装

▶ 本节视频教学录像：14 分钟

现在就来看一看面向对象的第一大特性——封装性。

10.1.1 封装的基本概念

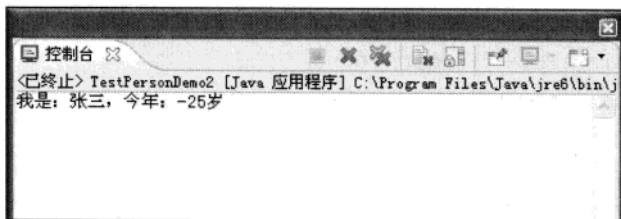
什么是封装性？读者可以先看下面的程序，看看会产生什么问题。

【范例 10-1】 类的封装性使用范例 1（代码 10-1. java）。

```
01  class Person
02  {
03      String name ;
04      int age ;
05      void talk()
06      {
07          System.out.println("我是: "+name+", 今年: "+age+"岁");
08      }
09  }
10
11  public class TestPersonDemo2
12  {
13      public static void main(String[] args)
14      {
15          // 声明并实例化一个 Person 对象 p
16          Person p = new Person();
17          // 给 p 中的属性赋值
18          p.name = "张三";
19          // 在这里将对象 p 的年龄属性赋值为-25
20          p.age = -25;
21          // 调用 Person 类中的 talk()方法
22          p.talk();
23      }
24  }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~9 行声明了一个新的类 Person，类中有 name、age 两个属性，还有一个 talk() 方法用于输出信息。

第 16 行声明并实例化了一个 Person 的对象 p。

第 18~22 行分别为 p 对象中的属性赋值，并调用 talk() 方法。

【范例分析】

从程序中可以看到，第 20 行将年龄 (age) 赋值为 -25，这明显是一个不合法的数据，最终程序在调用 talk() 方法的时候会打印出错误的信息。这好比要加工一件产品一样，加工的原料本身就有问题，那么最终加工出来的产品也一定是一个不合格的产品。而导致这种错误的原因，就是因为程序在原料的入口处没有检验，而加工的原料原本就是变质的，这样加工出来的产品也必然是一个不合要求的产品。

10.1.2 类的封装实例

读者可以发现，之前所列举的程序都是用对象直接访问类中的属性，这在面向对象法则中是不允许的。所以为了避免程序中这种错误的发生，在一般的开发中往往要将类中的属性封装 (private)。对范例 TestPersonDemo2.java 做了相应的修改后，就可构成下面的程序 TestPersonDemo3-1.java。

【范例 10-2】 类的封装性使用范例 2 (代码 10-2.java)。

```

01  class Person
02  {
03      private String name ;
04      private int age ;
05      void talk()
06      {
07          System.out.println("我是: "+name+", 今年: "+age+"岁");
08      }
09  }
10
11  public class TestPersonDemo3_1
12  {
13      public static void main(String[] args)

```



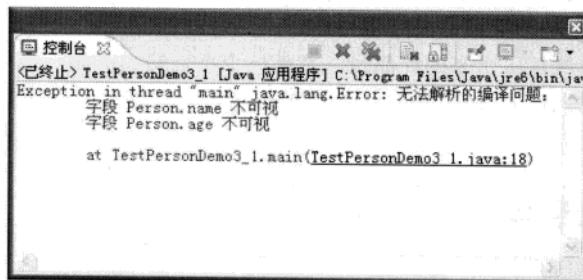
```

14  {
15      // 声明并实例化一个 Person 对象 p
16      Person p = new Person();
17      // 给 p 中的属性赋值
18      p.name = "张三";
19      // 在这里将 p 对象中的年龄赋值为-25
20      p.age = -25;
21      // 调用 Person 类中的 talk()方法
22      p.talk();
23  }
24  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~9 行声明了一个新的类 Person，类中有 name、age 两个属性，还有一个 talk()方法用于输出信息。与前面不同的是，这里的属性在声明时前面都加上了 private 关键字。

第 16 行声明并实例化了一个 Person 类的对象 p。

第 18~22 行分别为 p 对象中的属性赋值，并调用 talk()方法。

【范例分析】

可以看到本程序与上面的范例除了在声明属性上有些区别之外，并没有其他的区别。而就是这个小小的关键字，却使得程序连编译都无法通过，所提示的错误为：属性（name、age）为私有的，所以不能由对象直接进行访问，这样就可以保证对象无法直接去访问类中的属性。但是如果非要给对象赋值的话，这一矛盾该如何解决呢？程序设计人员一般在类的设计时，会对属性增加一些方法，如 setXxx()、getXxx()这样的公有方法来解决这一矛盾。请看下面的范例。

【范例 10-3】 类的封装性使用范例 3（代码 10-3.java）。

```

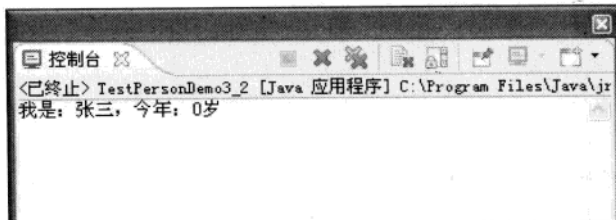
01  class Person
02  {
03      private String name ;
04      private int age ;

```

```
05     void talk()
06     {
07         System.out.println("我是: "+name+", 今年: "+age+"岁");
08     }
09     public void setName(String str)
10     {
11         name = str ;
12     }
13     public void setAge(int a)
14     {
15         if(a>0)
16             age = a ;
17     }
18     public String getName()
19     {
20         return name ;
21     }
22     public int getAge()
23     {
24         return age ;
25     }
26 }
27
28 public class TestPersonDemo3-2
29 {
30     public static void main(String[] args)
31     {
32         // 声明并实例化一个 Person 对象 p
33         Person p = new Person() ;
34         // 给 p 中的属性赋值
35         p.setName("张三") ;
36         // 在这里将 p 对象中的年龄赋值为-25
37         p.setAge(-25) ;
38         // 调用 Person 类中的 talk()方法
39         p.talk() ;
40     }
41 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 9~25 行加入了一些 `setXxx()`、`getXxx()` 方法，主要用来设置和取得类中的私有属性。

第 35 行调用了 `Person` 类中的 `setName()` 方法，并赋值为“张三”。第 37 行调用了 `setAge()` 方法，同时传进一个 -25 的不合理年龄。

第 13~17 行设置年龄的时候在程序中加了些判断语句，如果传入的数值大于 0，则将值赋给 `age` 属性。

【范例分析】

可以看到在本程序中，由于传进了一个 -25 的不合理的数值，这样在设置 `Person` 中属性的时候因为不满足条件而不能被设置成功，所以 `age` 的值依然为自己的默认值——0。这样在输出的时候可以看到，那些错误的数据并没有被赋到属性上去，而只输出了默认值。

由此可以发现，用 `private` 可以将属性封装起来，当然用 `private` 也可以封装方法，封装的形式如下。

封装属性：`private` 属性类型 属性名

封装方法：`private` 方法返回类型 方法名称 (参数)



注意：用 `private` 声明的属性或方法只能在其类的内部被调用，而不能在类的外部被调用。读者可以先暂时简单地理解为：在类的外部不能用对象去调用 `private` 声明的属性或方法。

下面的这个范例修改自上面的程序，在这里将 `talk()` 方法封装了起来。

【范例 10-4】 方法的封装使用 (代码 10-4.java)。

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      private void talk()
06      {
07          System.out.println("我是: "+name+", 今年: "+age+"岁");
08      }
09      public void setName(String str)
```

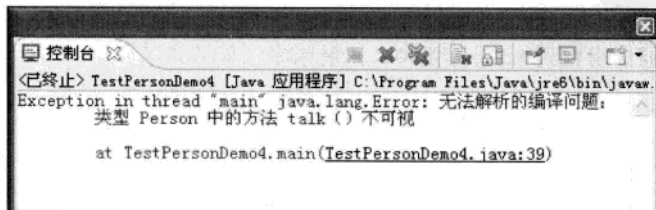
```

10     {
11         name = str ;
12     }
13     public void setAge(int a)
14     {
15         if(a>0)
16             age = a ;
17     }
18     public String getName()
19     {
20         return name ;
21     }
22     public int getAge()
23     {
24         return age ;
25     }
26 }
27
28 public class TestPersonDemo4
29 {
30     public static void main(String[] args)
31     {
32         // 声明并实例化一个 Person 对象 p
33         Person p = new Person() ;
34         // 给 p 中的属性赋值
35         p.setName("张三") ;
36         // 在这里将 p 对象中的年龄赋值为-25
37         p.setAge(-25) ;
38         // 调用 Person 类中的 talk()方法
39         p.talk() ;
40     }
41 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 9~25 行加入了一些 setXxx()、getXxx() 方法，主要用来设置和取得类中的私有属性。

第 35 行调用了类 Person 中的 setName() 方法，并赋值为“张三”。第 37 行调用了 setAge() 方法，同时传进一个不合理的年龄-25。

第 13~17 行设置年龄的时候在程序中加了些判断语句，如果传入的数值大于 0，则将值赋给 age 属性。

第 5 行将 talk() 方法用 private 来声明。

【范例分析】

可以看到 private 也是同样可以用来声明方法的，这样这个方法就只能在类的内部被访问了。



提示：读者可能会问，到底什么时候需要封装，什么时候不用封装。在这里可以告诉读者，关于封装与否并没有一个明确的规定，不过从程序设计的角度来说，一般说来设计较好的程序的类中的属性都是需要封装的。此时要设置或取得属性值，则只能使用 setXxx()、getXxx() 方法，这是一个明确且标准的规定。

10.2 类的继承

▶ 本节视频教学录像：17 分钟

在前面我们已经了解了类的基本使用方法。对于面向对象的程序而言，它的精华在于类的继承可以以既有的类为基础，进而派生出新的类。通过这种方式，便能快速地开发出新的类，而不需编写相同的程序代码，这就是程序代码再利用的概念。

10.2.1 继承的基本概念

在 Java 中，通过继承可以简化类的定义，扩展类的功能。在 Java 中支持类的单继承和多层继承，但是不支持多继承，即一个类只能继承一个类而不能继承多个类。

实现继承的格式如下。

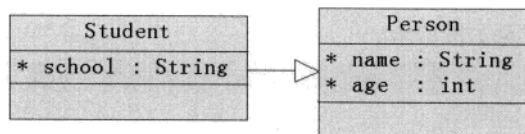
```
class 子类名 extends 父类
```

Java 继承只能直接继承父类中的公有属性和公有方法，而隐含地(不可见地)继承了私有属性。

现在假设有一个 Person 类，里面有 name 与 age 两个属性，而另外一个 Student 类，需要有 name、age、school 等 3 个属性，如图所示。在此可以看到 Person 中已经存在有 name 和 age 两个属性，所以不希望在 Student 类中再重新声明这两个属性，这个时候就需要考虑是不是可以将 Person 类中的内容继续保留到 Student 类中，这就引出了接下来要介绍的类的继承概念。

Student	Person
* school : String	* name : String
	* age : int

在这里希望 Student 类能够将 Person 类的内容继承下来后继续使用，可用下图表示。



Java 类的继承可用下面的语法来表示。

```

class 父类                                // 定义父类
{
}
class 子类 extends 父类                  // 用 extends 关键字实现类的继承
{
}
  
```

10.2.2 类的继承实例

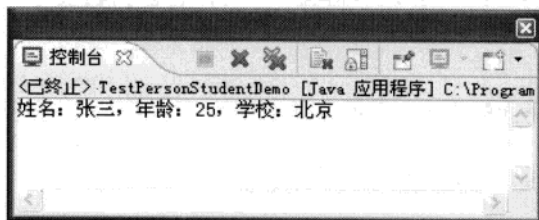
【范例 10-5】 类的继承演示程序（代码 10-5. java）。

```

01  class Person
02  {
03      String name ;
04      int age ;
05  }
06  class Student extends Person
07  {
08      String school ;
09  }
10
11  public class TestPersonStudentDemo
12  {
13      public static void main(String[] args)
14      {
15          Student s = new Student() ;
16          // 访问 Person 类中的 name 属性
17          s.name = "张三" ;
18          // 访问 Person 类中的 age 属性
19          s.age = 25 ;
20          // 访问 Student 类中的 school 属性
21          s.school = "北京" ;
22          System.out.println("姓名: "+s.name+" , 年龄: "+s.age+" , 学校: "+s.school) ;
23      }
24  }
  
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

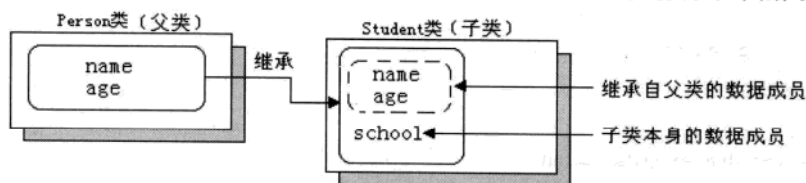
第 1~5 行声明了一个名为 Person 的类，里面有 name 和 age 两个属性。

第 6~9 行声明了一个名为 Student 的类，并继承自 Person 类。

第 15 行声明并实例化了一个 Student 类的对象 s。

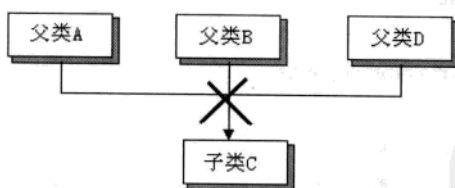
第 17、19、21 行分别用 Student 类的对象调用程序中的 name、age、school 属性。

从程序中可以看到，在 Student 类中虽然并未定义 name 与 age 属性，但在程序外部却依然可以调用 name 或 age。这是因为 Student 类直接继承自 Person 类，也就是说 Student 类直接继承了 Person 类中的属性，所以 Student 类的对象才可以访问到父类中的成员。如图所示。



提示：在 Java 中只允许单继承，而不允许多重继承，也就是说一个子类只能有一个父类。但在 Java 中却允许多层继承。

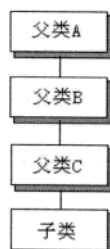
1. 多重继承



```
class A
{
}
class B
{
}
class C extends A,B
{
}
```

从代码中可以看到，类 C 同时继承了类 A 与类 B，也就是说类 C 同时继承了两个父类，这是不允许的。

2. 多层继承



```
class A
{
class B extends A
{
class C extends B
{
}
```

从代码中可以看到，类 B 继承了类 A，而类 C 又继承了类 B，也就是说类 B 是类 A 的子类，而类 C 则是类 A 的孙子类。

10.3 类的继承专题研究

▶ 本节视频教学录像：35 分钟

关于继承的问题，有一些概念和过程需要澄清，有些语法和术语需要熟练掌握，本节做一个总结。

10.3.1 子类对象的实例化过程

既然子类可以继承直接父类中的方法与属性，那么父类中的构造方法呢？请看下面的范例。

【范例 10-6】 子类对象的实例化（代码 10-6. java）。

```
01 class Person
02 {
03     String name ;
04     int age ;
05     // 父类的构造方法
06     public Person()
07     {
08         System.out.println("1.public Person(){}");
09     }
10 }
11 class Student extends Person
```

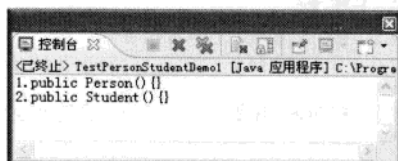
```

12 {
13     String school ;
14     // 子类的构造方法
15     public Student()
16     {
17         System.out.println("2.public Student(){}");
18     }
19 }
20
21 public class TestPersonStudentDemo1
22 {
23     public static void main(String[] args)
24     {
25         Student s = new Student() ;
26     }
27 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~10 行声明了一个 Person 类，此类中有一个无参构造方法。

第 11~19 行声明了一个 Student 类，此类继承自 Person 类，也有一个无参构造方法。

第 25 行声明并实例化了一个 Student 类的对象 s。

从程序输出结果中可以看到，虽然第 25 行实例化的是子类的对象，但是程序却先去调用父类中的无参构造方法，之后再调用子类本身的构造方法。所以由此可以得出结论：子类对象在实例化时会默认先去调用父类中的无参构造方法，之后再调用本类中的相应构造方法。

实际上在本范例中，在子类构造方法的第 1 行就默认隐含了一个“super()”语句。上面的程序如果改写成下面的形式，也是可以的。

```

class Student extends Person
{
    String school ;
    // 子类的构造方法
    public Student()
    {
        super() ;
        //实际上程序在这里隐含了这样一条语句

```



```

        System.out.println("2.public Student(){}");
    }
}

```

10.3.2 super 关键字的使用

在以前的程序中曾经提到过 super 的使用,那 super 到底是什么?从 TestPersonStudentDemo1 中读者应该可以发现,super 关键字出现在子类中,而且是去调用了父类中的构造方法,所以可以得出结论:super 主要的功能是完成子类调用父类中的内容,也就是调用父类中的属性或方法。将程序 TestPersonStudentDemo1.java 作相应的修改就构成了下面的范例 TestPersonStudentDemo2.java。

【范例 10-7】 super 调用父类中的构造方法 (代码 10-7. java)。

```

01  class Person
02  {
03      String name ;
04      int age ;
05      // 父类的构造方法
06      public Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;
10      }
11  }
12  class Student extends Person
13  {
14      String school ;
15      // 子类的构造方法
16      public Student()
17      {
18          // 在这里用 super 调用父类中的构造方法
19          super("张三",25);
20      }
21  }
22
23  public class TestPersonStudentDemo2
24  {
25      public static void main(String[] args)
26      {
27          Student s = new Student() ;

```

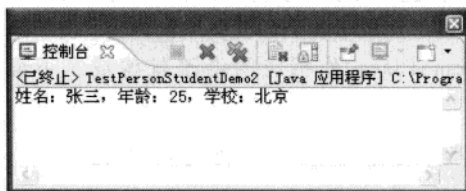
```

28      // 为 Student 类中的 school 赋值
29      s.school = "北京";
30      System.out.println("姓名: "+s.name+", 年龄: "+s.age+", 学校: "+s.school);
31  }
32  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~11 行声明了一个名为 Person 的类，里面有 name 和 age 两个属性，并声明了一个含有两个参数的构造方法。

第 12~21 行声明了一个名为 Student 的类，此类继承自 Person 类。第 16~20 行声明了一个子类的构造方法，在此方法中用 super("张三",25)调用父类中有两个参数的构造方法。

第 27 行声明并实例化了一个 Student 类的对象 s，第 29 行为 Student 对象 s 中的 school 赋值为“北京”。

读者可以看到，本例与范例 TestPersonStudentDemo3.java 的程序基本上是一样的，唯一的不同是在子类的构造方法中明确地指明了调用的是父类中有两个参数的构造方法，所以程序在编译时不再去找父类中无参的构造方法。



注 意：用 super 调用父类中的构造方法，只能放在程序的第 1 行。

super 关键字不仅可用于调用父类中的构造方法，也可用于调用父类中的属性或方法，例如下面的格式。

```

super.父类中的属性 ;
super.父类中的方法() ;输出结果:

```

【范例 10-8】 通过 super 调用父类的属性和方法（代码 10-8. java）。

```

01  class Person
02  {
03      String name ;
04      int age ;
05      // 父类的构造方法
06      public Person()

```

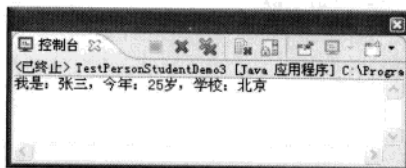
```

07     {
08     }
09     public String talk()
10     {
11         return "我是: "+this.name+", 今年: "+this.age+"岁";
12     }
13 }
14 class Student extends Person
15 {
16     String school;
17     // 子类的构造方法
18     public Student(String name,int age,String school)
19     {
20         // 在这里用 super 调用父类中的属性
21         super.name = name;
22         super.age = age;
23
24         // 调用父类中的 talk()方法
25         System.out.print(super.talk());
26
27         // 调用本类中的 school 属性
28         this.school = school;
29     }
30 }
31
32 public class TestPersonStudentDemo3
33 {
34     public static void main(String[] args)
35     {
36         Student s = new Student("张三",25,"北京");
37         System.out.println("姓名: "+s.name+", 年龄: "+s.age+", 学校: "+s.school);
38     }
39 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~13 行声明了一个名为 `Person` 的类，并声明了 `name` 和 `age` 两个属性、一个返回 `String` 类型的 `talk()` 方法，以及一个无参构造方法。

第 14~30 行声明了一个名为 `Student` 的类，此类直接继承自 `Person` 类。

第 21、22 行，通过 `super` 属性的方式调用父类中的 `name` 和 `age` 属性，并分别赋值。

第 25 行调用父类中的 `talk()` 方法打印信息。

从程序中可以看到，子类 `Student` 可以通过 `super` 调用父类中的属性或方法。但是细心的读者在本例中可以发现，如果第 21、22、25 行换成 `this` 调用也是可以的，那为什么还要用 `super` 呢？读者看完下面的内容就可以知道什么时候应该这样使用。

10.3.3 限制子类的访问

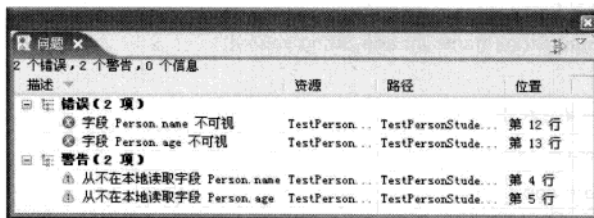
有些时候，父类并不希望子类可以访问自己的类中全部的属性或方法，所以需要将一些属性与方法隐藏起来，不让子类去使用。为此可在声明属性或方法时加上“`private`”关键字，表示私有。

【范例 10-9】 限制子类的访问（代码 10-9. java）。

```
01  class Person
02  {
03      // 在这里将属性封装
04      private String name ;
05      private int age ;
06  }
07  class Student extends Person
08  {
09      // 在这里访问父类中被封装的属性
10      public void setVar()
11      {
12          name = "张三";
13          age = 25 ;
14      }
15  }
16
17  class TestPersonStudentDemo4
18  {
19      public static void main(String[] args)
20      {
21          new Student().setVar();
22      }
23  }
```

【运行结果】

保存并运行程序，结果如图所示。



从编译器的错误提示可以看到，对 name 和 age 属性，在子类中无法进行访问。

10.3.4 覆写

“覆写”的概念与“重载”相似，它们均是 Java “多态”的技术之一。所谓“重载”，即是方法名称相同，但却可在不同的场合做不同的事。当一个子类继承一个父类，而子类中的方法与父类中的方法的名称、参数个数、类型等都完全一致时，就称子类中的这个方法覆写了父类中的方法。同理，如果子类中重复定义了父类中已有的属性，则称此子类中的属性覆写了父类中的属性。

```
class Super
{
    访问权限 方法返回值类型 方法 1 (参数 1)
    {}
}

class Sub extends Super
{
    访问权限 方法返回值类型 方法 1 (参数 1)    覆写父类中的方法
    {}
}
```

【范例 10-10】 子类覆写父类的实现（代码 10-10.java）。

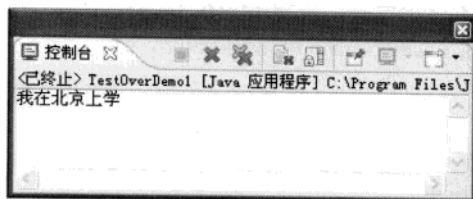
```
01 class Person
02 {
03     String name ;
04     int age ;
05     public String talk()
06     {
07         return "我是: "+this.name+", 今年: "+this.age+"岁";
08     }
09 }
10
```



```
11 class Student extends Person
12 {
13     String school ;
14     public Student(String name,int age,String school)
15     {
16         // 分别为属性赋值
17         this.name = name ;
18         this.age = age ;
19         this.school = school ;
20     }
21     // 此处覆写 Person 中的 talk()方法
22     public String talk()
23     {
24         return "我在"+this.school+"上学" ;
25     }
26 }
27
28 class TestOverDemo1
29 {
30     public static void main(String[] args)
31     {
32         Student s = new Student("张三",25,"北京") ;
33         // 此时调用的是子类中的 talk()方法
34         System.out.println(s.talk()) ;
35     }
36 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~9 行声明了一个名为 Person 的类，里面声明了 name 和 age 两个属性，和一个 talk() 方法。

第 11~26 行声明了一个名为 Student 的类，此类继承自 Person 类，也就继承了 name 和 age 属性，同时声明了一个与父类中同名的 talk() 方法，也可以说此时 Student 类中的 talk() 方法覆

写了 Person 类中的 talk() 方法。

第 32 行实例化了一个子类对象，并同时调用子类构造方法为属性赋初值。

第 34 行用子类对象调用 talk() 方法，但此时调用的是子类中的 talk() 方法。

从输出结果可以看到，在子类中覆写了父类中的 talk() 方法，所以子类对象在调用 talk() 方法时，实际上调用的是子类中被覆写好了的方法。另外可以看到，子类的 talk() 方法与父类的 talk() 方法在声明权限时，都声明为 public，也就是说这两个方法的访问权限都是一样的。

从 TestOverDemo1.java 程序中可以看到，第 34 行调用 talk() 方法实际上调用的只是子类的方法，那如果现在需要调用父类中的方法该如何实现呢？请看下面的范例，此范例修改自上一个范例。

【范例 10-11】 子类覆写父类的实现 2（代码 10-11.java）。

```
01 class Person
02 {
03     String name ;
04     int age ;
05     public String talk()
06     {
07         return "我是: "+this.name+"，今年: "+this.age+"岁";
08     }
09 }
10
11 class Student extends Person
12 {
13     String school ;
14     public Student(String name,int age,String school)
15     {
16         // 分别为属性赋值
17         this.name = name ;
18         this.age = age ;
19         this.school = school ;
20     }
21     // 此处覆写 Person 类中的 talk()方法
22     public String talk()
23     {
24         return super.talk()+"，我在"+this.school+"上学";
25     }
26 }
27
28 class TestOverDemo2
29 {
```

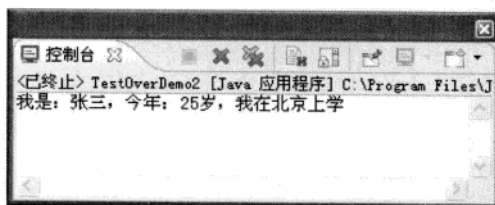
```

30     public static void main(String[] args)
31     {
32         Student s = new Student("张三",25,"北京");
33         //此时调用的是子类中的 talk()方法
34         System.out.println(s.talk());
35     }
36 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~9 行声明了一个 Person 类，里面声明了 name 和 age 两个属性，和一个 talk() 方法。

第 11~26 行声明了一个 Student 类，此类继承自 Person，也就继承了 name 和 age 属性，同时声明了一个与父类中同名的 talk() 方法，也可以说此时 Student 类中的 talk() 方法覆写了 Person 类中的 talk() 方法，但在第 24 行通过 super.talk() 方式，调用了父类中的 talk() 方法。

第 32 行实例化了一个子类对象，并同时调用子类构造方法为属性赋初值。

第 34 行用子类对象调用 talk() 方法，但此时调用的是子类中的 talk() 方法。

从程序中可以看到，在子类中可以通过 super.方法() 调用父类中被子类覆写的方法。

10.4 类的多态

本节视频教学录像：32 分钟

在前面已经介绍了面向对象的封装性和继承性。下面就来看一下面向对象中的最后一个，也是最重要的一个特性——多态性。

10.4.1 多态的基本概念

什么是多态性？读者应该还清楚在之前曾解释过重载的概念，重载的最终效果就是调用同一个方法名称，却可以根据传入参数的不同而得到不同的处理结果，这其实就是多态性的一种体现。

下面用一个范例简单地介绍一下多态的概念。

【范例 10-12】 对象多态性的使用（代码 10-12. java）。

```

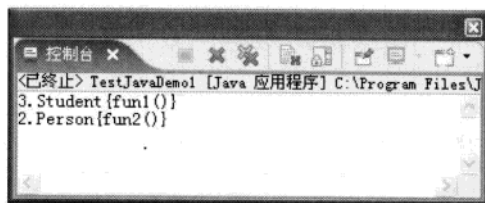
01  class Person

```

```
02 {
03     public void fun1()
04     {
05         System.out.println("1.Person{fun1()}");
06     }
07     public void fun2()
08     {
09         System.out.println("2.Person{fun2()}");
10     }
11 }
12
13 // Student 类扩展自 Person 类，也就继承了 Person 类中的 fun1()、fun2()方法
14 class Student extends Person
15 {
16     // 在这里覆写了 Person 类中的 fun1()方法
17     public void fun1()
18     {
19         System.out.println("3.Student{fun1()}");
20     }
21     public void fun3()
22     {
23         System.out.println("4.Studen{fun3()}");
24     }
25 }
26
27 class TestJavaDemo1
28 {
29     public static void main(String[] args)
30     {
31         // 此处，父类对象由于类实例化
32         Person p = new Student();
33         // 调用 fun1()方法，观察此处调用的是哪个类里的 fun1()方法
34         p.fun1();
35         p.fun2();
36     }
37 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~11 行声明了一个 Person 类，此类中有 fun1()、fun2() 两个方法。

第 14~25 行声明了一个 Student 类，此类继承自 Person 类，并覆写了 fun1() 方法。

第 32 行声明了一个 Person 类（父类）的对象 p，之后由子类对象去实例化此对象。

第 34 行由父类对象调用 fun1() 方法。

从程序的输出结果中可以看到，p 是父类的对象，但调用 fun1() 方法的时候并没有调用其本身的 fun1() 方法，而是调用了子类中被覆写的 fun1() 方法。之所以会产生这样的结果，最根本的原因就是因为父类对象并非由其本身的类实例化，而是通过子类实例化，这就是所谓的对象的多态性，即子类实例化对象可以转换为父类实例化对象。

10.4.2 类的多态实例

在这里要着重讲解两个概念，希望读者予以重视。

1. 向上转型

在上面的范例 TestJavaDemo1.java 中，父类对象通过子类对象去实例化，实际上就是对象的向上转型。向上转型是不需要进行强制类型转换的，但是向上转型会丢失精度。

2. 向下转型

与向上转型对应的一个概念就是“向下转型”，所谓向下转型，也就是说父类的对象可以转换为子类对象，但是需要注意的是，这时则必须要进行强制的类型转换。

读者可能觉得上面的两个概念有些难以理解，下面举个例子来帮助读者理解。

一天，有个小孩在马路上看见了一辆跑车，他指着跑车说那是汽车。相信读者都会认为这句话没有错，跑车的确是符合汽车的标准，所以把跑车说成汽车并没有错误，只是不准确而已。不管是小轿车也好，货车也好，其实都是汽车，这在现实生活中是说得通的。在这里读者可以将这些小轿车、货车都想象成汽车的子类，它们都是扩展了汽车的功能，都具备了汽车的功能，所以它们都可以叫做汽车，那么这种概念就称为向上转型。而相反，假如说把所有的汽车都当成跑车，那结果肯定是不正确的，因为汽车有很多种，必须明确地指明是哪辆跑车才可以，需要加一些限制，这个时候就必须明确地指明是哪辆车，所以需要进行强制的说明。

上面的解释可以概括成下面的两句话。

- (1) 向上转型可以自动完成。
- (2) 向下转型必须进行强制类型转换。



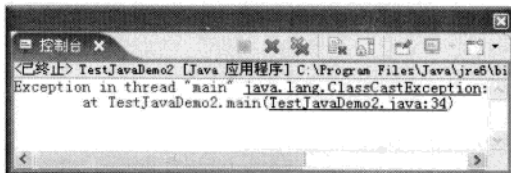
提示：另外要提醒读者注意的是，并非全部的父类对象都可以强制转换为子类对象，请看下面的范例。

【范例 10-13】 父类对象强制转换为子类对象使用实例（代码 10-13. java）。

```
01 class Person
02 {
03     public void fun1()
04     {
05         System.out.println("1.Person{fun1()}");
06     }
07     public void fun2()
08     {
09         System.out.println("2.Person{fun2()}");
10     }
11 }
12
13 // Student 类继承 Person 类，也就继承了 Person 类的 fun1()、fun2()方法
14 class Student extends Person
15 {
16     // 在这里覆写了 Person 类中的 fun1()方法
17     public void fun1()
18     {
19         System.out.println("3.Student{fun1()}");
20     }
21     public void fun3()
22     {
23         System.out.println("4.Studen{fun3()}");
24     }
25 }
26
27 class TestJavaDemo2
28 {
29     public static void main(String[] args)
30     {
31         // 此处，父类对象由自身实例化
32         Person p = new Person();
33         // 将 p 对象向下转型
34         Student s = (Student)p;
35         p.fun1();
36         p.fun2();
37     }
38 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

从程序中可以看到，第 32 行 Person 对象 p 是由 Person 类本身实例化的，在第 34 行将 Person 对象 p 强制转换为子类对象，这样写在语法上是没有任何错误的。但是在运行时可以发现出了异常，这是为什么？为什么父类不可以向子类转换了呢？其实这点并不难理解，读者可以想一下现实生活中的例子，假如你今天刚买完一些生活用品，回家的时候在路上碰见了一个孩子，这个孩子忽然对你说：“我是你的儿子，你把你的东西给我吧！”，这个时候你肯定不会把你的东西给这个孩子，因为你不确定他跟你是否有关系，怎么能给呢？那么在这程序中也是同样的道理，父类用其本身类实例化自己的对象，但它并不知道谁是自己的子类，那肯定在转换的时候会出现错误。那么这个错误该如何纠正呢？只需将第 32 行的代码修改成如下的形式即可。

```
Person p = new Student();
```

这个时候相当于是由子类去实例化父类对象，也就是说这个时候父类知道有这么一个子类，也就相当于父亲知道了自己有这么一个孩子，所以下面再进行转换的时候就不会再有问题了。

10.5 练一练

一、填空题

1. _____ 是类的一个重要属性，可以从一个简单的类继承出相对复杂高级的类。
2. 类的继承可以以既有的 _____ 为基础，进而派生出新的 _____。
3. 继承是 _____ 的一个重要属性。

二、简答题

1. 什么是重载？
2. 向上转型和向下转型各自的特点是什么？

10.6 跟我上机

请编写一个既有重载又有重写的程序。

第 11 章

抽象类与接口



本章视频教学录像：1 小时 27 分钟

Java可以创建一种类专门用来当作父类，这种类称为“抽象类”。抽象类的作用有点类似“模板”，其目的是要设计者依据它的格式来修改并创建新的类。本章讲述抽象类的基本概念以及接口的知识。

本章要点（已掌握的在方框中打勾）

- ☐ 了解抽象类的使用
- ☐ 掌握抽象类的基本概念
- ☐ 掌握抽象类实例的应用
- ☐ 掌握接口的基本概念
- ☐ 熟悉接口实例的应用



11.1 抽象类的基本概念

▶ 本节视频教学录像：13 分钟

在 Java 中可以创建一种类专门用来当做父类，这种类称为“抽象类”。抽象类实际上也是一个类，只是与之前的普通类相比，其中多了抽象方法。

抽象方法是只声明而未实现的方法，所有的抽象方法必须使用 `abstract` 关键字声明，包含抽象方法的类也必须使用 `abstract class` 声明。

抽象类定义规则如下。

- (1) 抽象类和抽象方法都必须用 `abstract` 关键字来修饰。
- (2) 抽象类不能被直接实例化，也就是不能直接用 `new` 关键字去产生对象。
- (3) 抽象方法只需声明，而不需实现。
- (4) 含有抽象方法的类必须被声明为抽象类，抽象类的子类必须覆写所有的抽象方法后才能被实例化，否则这个子类还是个抽象类。

```
abstract class 类名称           // 定义抽象类
{
    声明数据成员 ;

    访问权限 返回值的数据类型 方法名称 ( 参数... )
    {
        ...定义一般方法
    }
    abstract 返回值的数据类型 方法名称 ( 参数... );
    // 定义抽象方法，在抽象方法里没有定义方法体
}
```



注 意：在抽象类定义的语法中，方法的定义可分为两种：一种是一般的方法，它和先前介绍过的方法没有什么两样；另一种是“抽象方法”，它是以 `abstract` 关键字为开头的方法，此方法只声明了返回值的数据类型、方法名称与所需的参数，但没有定义方法体。

11.2 抽象类实例

▶ 本节视频教学录像：25 分钟

【范例 11-1】 限制子类的访问（代码 11-1. java）。

```
01  abstract class Person
02  {
```

```
03     String name ;
04     int age ;
05     String occupation ;
06     // 声明一抽象方法 talk()
07     public abstract String talk() ;
08 }
09 // Student 类继承自 Person 类
10 class Student extends Person
11 {
12     public Student(String name,int age,String occupation)
13     {
14         this.name = name ;
15         this.age = age ;
16         this.occupation = occupation ;
17     }
18     // 覆写 talk()方法
19     public String talk()
20     {
21         return "学生——>姓名: "+this.name+", 年龄: "+this.age+",
22         职业: "+this.occupation+"! " ;
23     }
24 }
25 // Worker 类继承自 Person 类
26 class Worker extends Person
27 {
28     public Worker(String name,int age,String occupation)
29     {
30         this.name = name ;
31         this.age = age ;
32         this.occupation = occupation ;
33     }
34     // 覆写 talk()方法
35     public String talk()
36     {
37         return "工人——>姓名: "+this.name+",
38         年龄: "+this.age+", 职业: "+this.occupation+"! " ;
39     }
40 }
41 class TestAbstractDemo1
42 {
43     public static void main(String[] args)
```



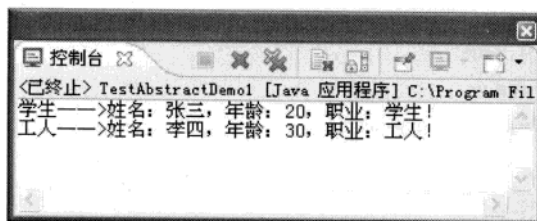
```

44     {
45         Student s = new Student("张三",20,"学生");
46         Worker w = new Worker("李四",30,"工人");
47         System.out.println(s.talk());
48         System.out.println(w.talk());
49     }
50 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~8 行声明了一个名为 `Person` 的抽象类，在 `Person` 中声明了 3 个属性和一个抽象方法——`talk()`。

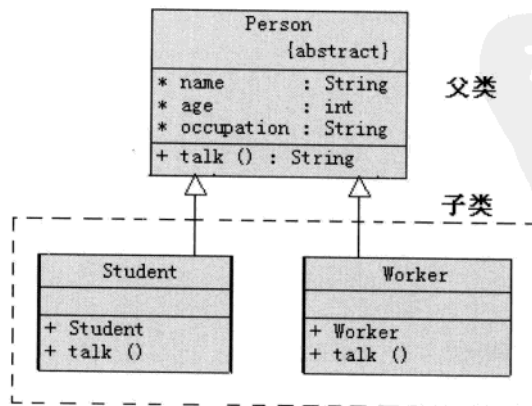
第 9~24 行声明了一个 `Student` 类，此类继承自 `Person` 类，而且此类不为抽象类，所以需要覆写 `Person` 类中的抽象方法——`talk()`。

第 25~40 行声明了一个 `Worker` 类，此类继承自 `Person` 类，而且此类不为抽象类，所以需要覆写 `Person` 类中的抽象方法——`talk()`。

第 45、46 行分别实例化 `Student` 类与 `Worker` 类的对象，并调用各自的构造方法初始化类属性。

第 47、48 行分别调用各自类中被覆写的 `talk()` 方法。

可以看到两个子类 `Student`、`Worker` 都分别按各自的要求覆写了 `talk()` 方法。上面的程序可由下图表示。





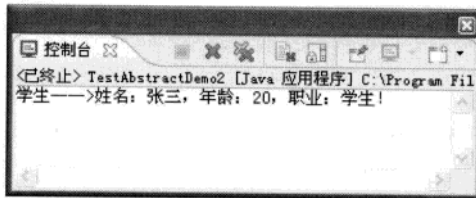
提示：与一般类相同，在抽象类中也可以拥有构造方法，但是这些构造方法必须在子类中被调用。

【范例 11-2】 抽象类的使用（代码 11-2. java）。

```
01  abstract class Person
02  {
03      String name ;
04      int age ;
05      String occupation ;
06      public Person(String name,int age,String occupation)
07      {
08          this.name = name ;
09          this.age = age ;
10          this.occupation = occupation ;
11      }
12      public abstract String talk() ;
13  }
14  class Student extends Person
15  {
16      public Student(String name,int age,String occupation)
17      {
18          // 在这里必须明确调用抽象类中的构造方法
19          super(name,age,occupation);
20      }
21      public String talk() {
22          return "学生——>姓名: "+this.name+", 年龄: "+this.age+", 职业: "+this.occupation+"! ";
23      }
24  }
25  class TestAbstractDemo2
26  {
27      public static void main(String[] args)
28      {
29          Student s = new Student("张三",20,"学生");
30          System.out.println(s.talk());
31      }
32  }
```

【运行结果】

保存并运行程序，结果如图所示。



从程序中可以看到, 抽象类也可以像普通类一样, 有构造方法、一般方法和属性, 更重要的是还可以有一些抽象方法, 留给子类去实现, 而且在抽象类中声明构造方法后, 在子类中必须明确调用。

11.3 接口的基本概念

本节视频教学录像: 13 分钟

接口 (interface) 是 Java 所提供的另一种重要技术, 它的结构和抽象类非常相似, 也具有数据成员与抽象方法, 但它与抽象类又有以下两点不同。

- (1) 接口里的数据成员必须初始化, 且数据成员均为常量。
- (2) 接口里的方法必须全部声明为 abstract, 也就是说, 接口不能像抽象类一样保有一般的方法, 必须全部是“抽象方法”。

接口定义的语法如下。

```
interface 接口名称                // 定义抽象类
{
    final 数据类型 成员名称 = 常量;    // 数据成员必须赋初值
    abstract 返回值的数据类型 方法名称 (参数...);
    // 抽象方法, 注意在抽象方法里没有定义方法主体
}
```

接口与一般类一样, 本身也具有数据成员与方法, 但数据成员一定要赋初值, 且此值不能再更改, 方法也必须是“抽象方法”。也正因为方法必须是抽象方法, 而没有一般的方法, 所以如上格式中, 抽象方法声明的关键字 abstract 是可以省略的。

相同的情况也发生在数据成员身上, 因数据成员必须赋初值, 且此值不能再被更改, 所以声明数据成员的关键字 final 也可省略。事实上只要记住以下两点即可。

- (1) 接口里的“抽象方法”只要做声明即可, 而不用定义其处理的方式。
- (2) 数据成员必须赋初值。

在 Java 中接口是用于实现多继承的一种机制, 也是 Java 设计中最重要的一环, 每一个由接口实现的类必须在类内部覆写接口中的抽象方法, 且可自由地使用接口中的常量。

既然接口里只有抽象方法, 它只要声明而不用定义处理方式, 于是自然可以联想到接口有没有办法像一般类一样, 再用它来创建对象。利用接口打造新的类的过程, 称之为接口的实现 (implementation)。

以下为接口实现的语法。

```
class 类名称 implements 接口 A, 接口 B    // 接口的实现
```

```
{  
    ...  
}
```

11.4 接口实例

▶ 本节视频教学录像：36 分钟

下面的范例修改自范例 TestAbstractDemo1.java。

【范例 11-3】 接口的使用（代码 11-3.java）。

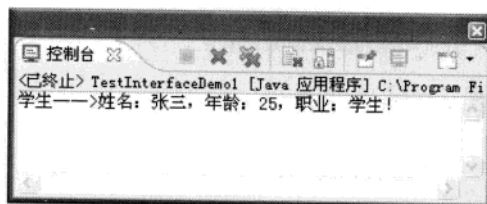
```
01 interface Person  
02 {  
03     String name = "张三";  
04     int age = 25 ;  
05     String occupation = "学生";  
06     // 声明一抽象方法 talk()  
07     public abstract String talk() ;  
08 }  
09 // Student 类实现自 Person 类  
10 class Student implements Person  
11 {  
12     // 覆写 talk()方法  
13     public String talk()  
14     {  
15         return "学生——>姓名: "+this.name+" , 年龄: "+this.age+" , 职业: "+this.occupation+"! ";  
16     }  
17 }  
18 class TestInterfaceDemo1  
19 {  
20     public static void main(String[] args)  
21     {
```

```

22      Student s = new Student();
23      System.out.println(s.talk());
24  }
25  }
    
```

【运行结果】

保存并运行程序，结果如图所示。



【代码注解】

第 1~8 行声明了一个 Person 接口，并在里面声明了 3 个常量：name、age、occupation，并分别赋值。

第 10~18 行声明了一个 Student 类，此类实现 Person 接口，并覆写 Person 中的 talk() 方法。

第 22 行实例化了一个 Student 的对象 s，并在第 23 行调用 talk() 方法打印信息。

有些读者可能会觉得这样做与抽象类并没有什么不同，在这里需要再次提醒读者的是：接口是 Java 实现多继承的一种机制，一个类只能继承一个父类，但如果需要一个类继承多个抽象方法的话，就明显无法实现，所以就出现了接口的概念。一个类只可以继承一个父类，但却可以实现多个接口。

接口与一般类一样，均可通过扩展的技术来派生出新的接口。原来的接口称为基本接口或父接口，派生出的接口称为派生接口或子接口。通过这种机制，派生接口不仅可以保留父接口的成员，同时也可加入新的成员以满足实际的需要。

同样，接口的扩展（或继承）也是通过关键字 extends 来实现的。有趣的是，一个接口可以继承多个接口，这点与类的继承有所不同。下面是接口扩展的语法。

```

interface 子接口名称 extends 父接口 1, 父接口 2, ...
{
    .....
}
    
```

【范例 11-4】 接口的继承使用（代码 11-4. java）。

```

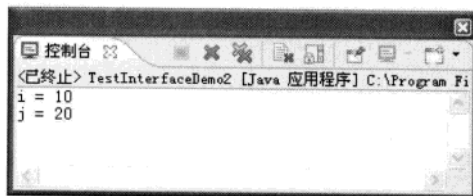
01  interface A
02  {
03      int i = 10;
04      public void sayI();
05  }
06  interface E
    
```



```
07 {
08     int x = 40 ;
09     public void sayE() ;
10 }
11 // B 同时继承了 A、E 两个接口
12 interface B extends A,E
13 {
14     int j = 20 ;
15     public void sayJ() ;
16 }
17
18 // C 继承实现 B 接口，也就意味着要实现 A、B、E 等 3 个接口的抽象方法
19 class C implements B
20 {
21     public void sayI()
22     {
23         System.out.println("i = "+i);
24     }
25     public void sayJ()
26     {
27         System.out.println("j = "+j) ;
28     }
29     public void sayE()
30     {
31         System.out.println("e = "+x);
32     }
33 }
34 class TestInterfaceDemo2
35 {
36     public static void main(String[] args)
37     {
38         C c = new C() ;
39         c.sayI() ;
40         c.sayJ() ;
41     }
42 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~5 行声明了一个接口 A，并声明了一个常量 i 和一个抽象方法 sayI()。

第 6~10 行声明了一个接口 E，并声明了一个常量 x 和一个抽象方法 sayE()。

第 12~16 行声明了一个接口 B，此接口同时继承 A、E 接口，同时声明了一个常量 j 和一个抽象方法 sayJ()。

第 19~33 行声明了一个类 C，此类实现了 B 接口，所以此类中要覆写接口 A、B、E 中的全部抽象方法。

从此程序中读者可以看到与类的继承不同的是：一个接口可以同时继承多个接口，也就是同时继承了多个接口的抽象方法与常量。

11.5 练一练

一、填空题

1. _____ 与一般类一样，本身也具有数据成员与方法，但数据成员一定要赋初值，且此值不能再更改，方法也必须是“抽象方法”。
2. 接口里的“抽象方法”只要做_____即可，而不用定义其处理的方式。
3. 接口与一般类一样，均可通过_____的技术来派生出新的接口。

二、简答题

1. 简述接口（interface）的概念。
2. 抽象类定义规则是什么？

11.6 跟我上机

设计一个限制子类的访问的抽象类实例，要求在控制台输出如下结果。

干部→姓名：李乐乐，年龄：20，职业：干部

工人→姓名：王小六，年龄：30，职业：工人

第 12 章

关于类的专题研究



本章视频教学录像：3 小时 26 分钟

Java中有一个比较特殊的类，就是Object类，它是所有类的父类，其功能非常强大。本章介绍Object类的概念、匿名内部类、匿名对象、类的使用方法、数据类型的传递、接口对象以及关键字的使用等知识和技能。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握 Object 类的基本概念
- ☐ 熟悉内部类的使用
- ☐ 了解匿名对象的使用
- ☐ 掌握方法的运用技巧
- ☐ 掌握关键字的作用与使用技巧



12.1 众类鼻祖——Object 类

▶ 本节视频教学录像：22 分钟

Java 中有一个比较特殊的类，就是 Object 类，它是所有类的父类。如果一个类没有使用 extends 关键字明确标识继承另外一个类，那么这个类就默认继承 Object 类。因此，Object 类是 Java 类层中的最高层类，是所有类的超类。换句话说，Java 中任何一个类都是它的子类。由于所有的类都是由 Object 类衍生出来的，所以 Object 类中的方法适用于所有类。

```
public class Person // 当没有指定父类时，会默认 Object 类为其父类
{
    ...
}
```

上面的程序等价于：

```
public class Person extends Object
{
    ...
}
```

读者查一下 JDK 的帮助文档，可以发现在 Object 类的方法中有一个 toString() 方法。

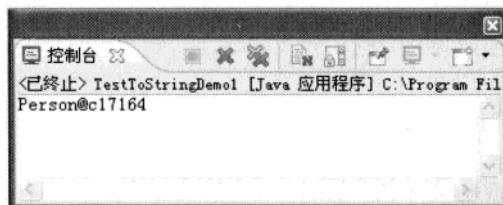
此方法是在打印对象时被调用的。下面有两个范例，一个是没有覆写 toString() 方法，另一个是覆写了 toString() 方法，读者可以比较两者的区别。

【范例 12-1】 Object 类的使用（代码 12-1. java）。

```
01 class Person extends Object
02 {
03     String name = "张三";
04     int age = 25 ;
05 }
06 class TestToStringDemo1
07 {
08     public static void main(String[] args)
09     {
10         Person p = new Person();
11         System.out.println(p);
12     }
13 }
```

【运行结果】

保存并运行程序，结果如图所示。

**【代码注解】**

第 1~5 行声明了一个名为 Person 的类，并明确指出其继承自 Object 类。

第 10 行声明并实例化了一个 Person 类的对象 p，第 11 行打印对象。

从程序中可以看到，在打印对象 p 的时候实际上打印出来的是一些无序的字符串，这样的字符串很少有人能看懂是什么意思。之后可以再观察下面的范例，范例覆写了 Object 类中的 toString() 方法。

【范例 12-2】 覆写 Object 类的方法（代码 12-2. java）。

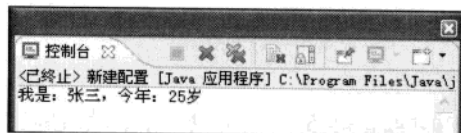
```

01  class Person extends Object
02  {
03      String name = "张三";
04      int age = 25 ;
05      // 覆写 Object 类中的 toString()方法
06      public String toString()
07      {
08          return "我是: "+this.name+"， 今年: "+this.age+"岁";
09      }
10  }
11  class TestToStringDemo2
12  {
13      public static void main(String[] args)
14      {
15          Person p = new Person();
16          System.out.println(p);
17      }
18  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

与 TestToStringDemo1.java 程序相比, 程序 TestToStringDemo2.java 程序在 Person 类中明确覆写了 toString() 方法, 这样在打印对象 p 的时候, 实际上是去调用 toString() 方法, 只是没有明显地指明调用 toString() 方法而已。此时第 16 行相当于:

```
System.out.println(p.toString());
```

12.2 内部类

本节视频教学录像: 17 分钟

现在已经知道, 在类内部可定义成员变量与方法, 有趣的是, 在类内部也可以定义另一个类。如果在类 Outer 的内部再定义一个类 Inner, 此时类 Inner 就称为内部类, 而类 Outer 则称为外部类。

内部类可声明成 public 或 private。当内部类声明成 public 或 private 时, 对其访问的限制与成员变量和成员方法完全相同。

内部类的定义格式如下。

```
标识符 class 外部类的名称
{
    // 外部类的成员
    标识符 class 内部类的名称
    {
        // 内部类的成员
    }
}
```

【范例 12-3】 内部类的使用 (代码 12-3. java)。

```
01 class Outer
02 {
03     int score = 95;
04     void inst()
05     {
06         Inner in = new Inner();
07         in.display();
08     }
09     class Inner
```

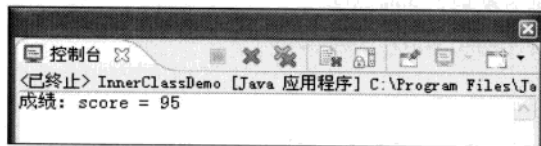
```

10    {
11        void display()
12        {
13            System.out.println("成绩: score = " + score);
14        }
15    }
16 }
17 public class InnerClassDemo
18 {
19     public static void main(String[] args)
20     {
21         Outer outer = new Outer();
22         outer.inst();
23     }
24 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 9~15 行，在 Outer 类的内部声明了一个 Inner 类，此类中有一个 display() 方法，用于打印外部类中的 score 属性。

第 4~8 行声明了一个 inst() 方法，用于实例化内部类的对象 in。

【范例分析】

从程序中可以看到，内部类 Inner 可以直接调用外部类 Outer 中的 score 属性，现在如果把内部类拿到外面来单独声明，那么在使用外部类中的 score 属性时，则需要先产生 Outer 类的对象，再由对象去调用 Outer 类的 score 属性。

可以看到由于使用了内部类操作，所以程序在调用 score 属性的时候减少了创建对象的操作，从而省去了一部分的内存开销。但是内部类在声明时，会破坏程序的结构，因此在开发中往往不建议读者去使用。

从程序中可以看到，外部类声明的属性可以被内部类所访问，那内部类声明的属性可不可以被外部类所访问？请看下面的范例。

【范例 12-4】 内部类声明的属性访问（代码 12-4. java）。

```

01 class Outer

```

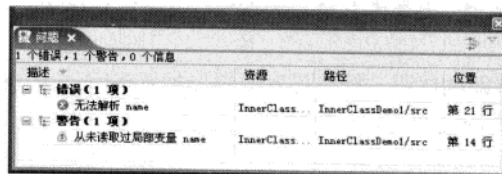
```

02  {
03      int score = 95;
04      void inst()
05      {
06          Inner in = new Inner();
07          in.display();
08      }
09      public class Inner
10      {
11          void display()
12          {
13              // 在内部类中声明一个 name 属性
14              String name = "张三";
15              System.out.println("成绩: score = " + score);
16          }
17      }
18      public void print()
19      {
20          // 在此调用内部类的 name 属性
21          System.out.println("姓名: " + name);
22      }
23  }
24  public class InnerClassDemo1
25  {
26      public static void main(String[] args)
27      {
28          Outer outer = new Outer();
29          outer.inst();
30      }
31  }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

从程序编译结果中可以看到，外部类无法找到内部类中所声明的属性，而内部类则可访问外

部类的属性。

前面已经学过 static 的用法, 用 static 可以声明属性或方法, 而用 static 也可以声明内部类, 用 static 声明的内部类则变成了外部类, 但是用 static 声明的内部类不能访问非 static 的外部类属性。

【范例 12-5】 用 static 声明的内部类访问非 static 的外部类属性 (代码 12-5. java)。

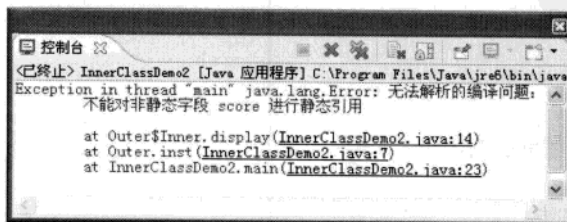
```

01  class Outer
02  {
03      int score = 95;
04      void inst()
05      {
06          Inner in = new Inner();
07          in.display();
08      }
09      // 这里用 static 声明内部类
10      static class Inner
11      {
12          void display()
13          {
14              System.out.println("成绩: score = " + score);
15          }
16      }
17  }
18  public class InnerClassDemo2
19  {
20      public static void main(String[] args)
21      {
22          Outer outer = new Outer();
23          outer.inst();
24      }
25  }

```

【运行结果】

保存并运行程序, 结果如图所示。



【范例分析】

由编译结果可以看到,由于内部类 Inner 声明为 static 类型,所以无法访问外部类中的非 static 类型属性 score。

12.2.1 在类外部引用内部类

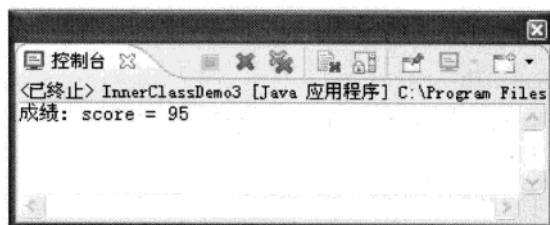
内部类也可以通过创建对象从外部类之外被调用,只要将内部类声明为 public 即可,请看下面的范例。

【范例 12-6】 在类的外部引用内部类(代码 12-6. java)。

```
01  class Outer
02  {
03      int score = 95;
04      void inst()
05      {
06          Inner in = new Inner();
07          in.display();
08      }
09      public class Inner
10      {
11          void display()
12          {
13              System.out.println("成绩: score = " + score);
14          }
15      }
16  }
17  public class InnerClassDemo3
18  {
19      public static void main(String[] args)
20      {
21          Outer outer = new Outer();
22          Outer.Inner inner = outer.new Inner();
23          inner.display();
24      }
25  }
```

【运行结果】

保存并运行程序,结果如图所示。



【代码详解】

第 9~15 行用 `public` 声明了一个内部类，此内部类可被外部类访问。

第 22 行用外部类的对象去实例化了一个内部类的对象。

12.2.2 在方法中定义内部类

内部类不仅可以在类中定义，也可以在方法中定义。

【范例 12-7】 在方法中定义内部类（代码 12-7. java）。

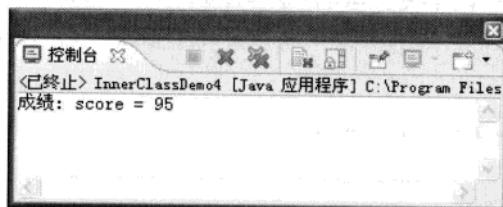
```

01  class Outer
02  {
03      int score = 95;
04      void inst()
05      {
06          class Inner
07          {
08              void display()
09              {
10                  System.out.println("成绩: score = " + score);
11              }
12          }
13          Inner in = new Inner();
14          in.display();
15      }
16  }
17  public class InnerClassDemo4
18  {
19      public static void main(String[] args)
20      {
21          Outer outer = new Outer();
22          outer.inst();
23      }
24  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 4~15 行在 Outer 类中声明了一个 inst()方法，在此方法中声明了一个 Inner 的内部类，同时产生了 Inner 的内部类实例化对象，调用其内部的方法。

第 21 行产生了一个 Outer 类的实例化对象，第 22 行调用 Outer 类中的 inst()方法。

在方法中定义的内部类只能访问方法中的 final 类型的局部变量，因为用 final 定义的局部变量相当于一个常量，它的生命周期超出方法运行的生命周期，如下面的范例。

【范例 12-8】 final 定义局部变量的生命周期 1（代码 12-8. java）。

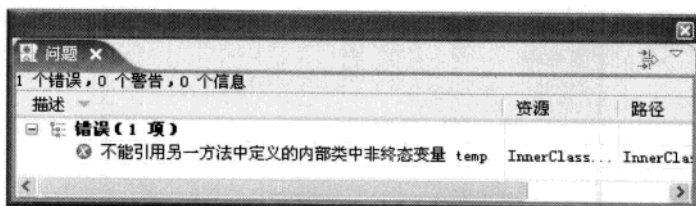
```

01  class Outer
02  {
03      int score = 95;
04      void inst(final int s)
05      {
06          int temp = 20 ;
07          class Inner
08          {
09              void display()
10              {
11                  System.out.println("成绩: score = " + (score+s+temp));
12              }
13          }
14          Inner in = new Inner();
15          in.display();
16      }
17  }
18  public class InnerClassDemo5
19  {
20      public static void main(String[] args)
21      {
22          Outer outer = new Outer();
23          outer.inst(5);
24      }
    
```

25 }

【运行结果】

保存并运行程序，结果如图所示。

**【范例分析】**

从编译结果可以看到，内部类可以访问用 final 标记的变量 s，却无法访问在方法内声明的变量 temp，所以只要将第 6 行的变量声明时，加上一个 final 修饰即可：final int temp = 20。完整程序如下。

【范例 12-9】 final 定义局部变量的生命周期 2（代码 12-9. java）。

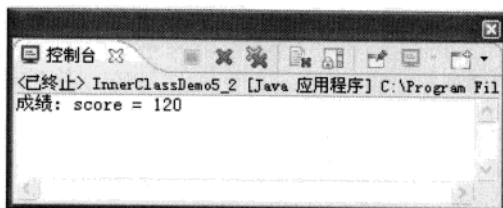
```

01  class Outer
02  {
03      int score = 95;
04      void inst(final int s)
05      {
06          final int temp = 20 ;
07          class Inner
08          {
09              void display()
10              {
11                  System.out.println("成绩: score = " + (score+s+temp));
12              }
13          }
14          Inner in = new Inner();
15          in.display();
16      }
17  }
18  public class InnerClassDemo5
19  {
20      public static void main(String[] args)
21      {
22          Outer outer = new Outer();
23          outer.inst(5);
24      }

```

【运行结果】

保存并运行程序，结果如图所示。

**【代码注解】**

第 4~16 行声明了一个 inst() 方法，并在此方法参数处声明了一个 final 类型的变量。

第 6 行声明了一个 final 类型的变量 temp，此变量用 final 声明，表示此变量可以由 Inner 类调用。

第 22 行声明并实例化了一个 Outer 类型的对象 outer。

outer 对象调用本类中的 inst() 方法。

12.3 匿名内部类

本节视频教学录像：7 分钟

在前面已经介绍过内部类的概念，而之前所介绍的内部类只是单一的类，并没有继承或者实现任何类或接口，实际上也是可以继承一个抽象类或实现一个接口。

下面先来看一个比较简单的范例。

【范例 12-10】 匿名内部类使用实例 1（代码 12-10. java）。

```
01 interface A
02 {
03     public void fun1();
04 }
05 class B
06 {
07     int i = 10;
08     class C implements A
09     {
10         public void fun1()
11         {
12             System.out.println(i);
13         }
14     }
```

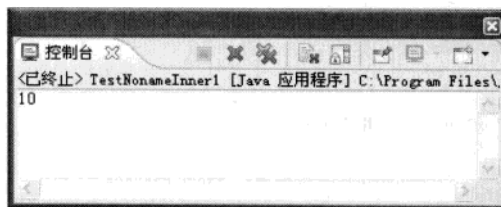
```

15     public void get(A a)
16     {
17         a.fun1();
18     }
19     public void test()
20     {
21         this.get(new C());
22     }
23 }
24 class TestNonameInner1
25 {
26     public static void main(String[] args)
27     {
28         B b = new B();
29         b.test();
30     }
31 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~4 行声明了一个 A 接口，并声明了一个 fun1() 抽象方法。

第 5~23 行声明了一个类 B，此类中有一个内部类 C 和一个变量 i。

第 8~14 行在类 B 中声明另一个内部类 C，此类实现 A 接口，并覆写了 fun1 方法。

第 15~18 行声明了一个 get() 方法，此方法用于 A 接口对象的实例化，并调用 fun1() 方法。

第 19~22 行声明了一个 test() 方法，此方法用于调用 get() 方法。

第 28 行声明了一个 B 类的实例化对象 b，并在第 29 行调用 test() 方法。

对上面的程序，有些读者可能会认为与以前的内部类没有太大的区别，也是在类内部声明了一个类，与原先不同的是多实现了一个接口罢了。的确，这里只是简单地声明了一个内部类，目的并不是要再重新介绍一遍内部类的概念，而是为了引出下面的概念——匿名内部类。

【范例 12-11】 匿名内部类使用实例 2 (代码 12-11.java)。

```

01 interface A
02 {

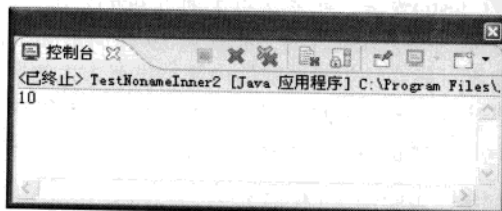
```



```
03     public void fun1() ;
04 }
05 class B
06 {
07     int i = 10 ;
08     public void get(A a)
09     {
10         a.fun1() ;
11     }
12     public void test()
13     {
14         this.get(new A())
15         {
16             public void fun1()
17             {
18                 System.out.println(i) ;
19             }
20         }
21     };
22 }
23 }
24 class TestNonameInner2
25 {
26     public static void main(String[] args)
27     {
28         B b = new B() ;
29         b.test() ;
30     }
31 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

由此程序可以看到，程序中并没有明确地声明出实现接口 A 的类，但是在第 14~21 行可

以看到，程序实现了接口 A 中的 fun1() 方法，并把整个的一个实现类传递到了方法 get() 中，这就是所谓的匿名内部类，它不用声明实质上的类就可以使用。对第 14~21 行有些读者会觉得难以理解，下面将这段程序拆分后再进行说明。

```
get(new A());
```

这段代码读者应该可以看明白，它的主要作用是传递一个 A 的匿名对象，之后实现 A 接口里的 fun1 方法。

```
get(new A()
{
})
);
get(new A()
{
    public void fun1()
    {
        System.out.println(i);
    }
})
);
```

上面的程序就是匿名内部类。有些读者可能会觉得难以理解，但是这样的程序见多了、用多了，也就能慢慢理解了。

12.4 匿名对象

▶ 本节视频教学录像：2 分钟

匿名对象，顾名思义就是没有明确的声明的对象。读者也可以简单地理解为只使用一次的对象，即没有任何一个具体的对象名称引用它。请看下面的范例。

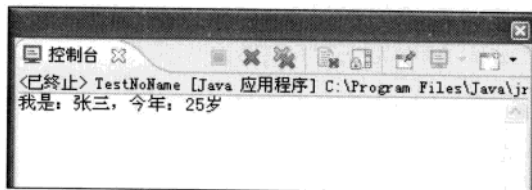
【范例 12-12】 匿名对象的使用（代码 12-12.java）。

```
01 class Person
02 {
03     private String name = "张三";
04     private int age = 25;
05     public String talk()
06     {
07         return "我是：" + name + "，今年：" + age + "岁";
08     }
09 }
10
```

```
11 public class TestNoName
12 {
13     public static void main(String[] args)
14     {
15         System.out.println(new Person().talk());
16     }
17 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~9 行声明了一个 Person 类，里面有 name 和 age 两个私有属性，并分别赋了初值。

第 15 行声明了一个 Person 匿名对象，调用 Person 类中的 talk() 方法。

【范例分析】

从程序中可以看到，用 new Person() 声明的对象并没有赋给任何一个 Person 类对象的引用，所以此对象只使用了一次，之后就会被 Java 的垃圾收集器回收。

12.5 再谈方法

▶ 本节视频教学录像：1 小时 7 分钟

方法可以简化程序的结构，也可以节省编写相同程序代码的时间，达到程序模块化的目的。

其实读者对方法应该不陌生，在每一个类里出现的 main() 即是一个方法。使用方法来编写程序代码有相当多的好处，它可以简化程序代码，精简重复的程序流程，并可把具有特定功能的程序代码独立出来，使程序的维护成本降低。

方法可用如下的语法来定义。

```
返回值类型 方法名称 ( 类型 参数 1, 类型 参数 2, ... )
{
    程序语句 ;
    return 表达式;
}
```

需要特别注意的是：如果不需要传递参数到方法中，只要将括号写出即可，不必填入任何内容。此外，如果方法没有返回值，return 语句则可省略。

TestJava4_8 是一个简单的方法操作范例，它在显示器上先输出 19 个星号 “*”，换行之后再输出 “I Like Java!” 这一字符串，最后再输出 19 个星号。

【范例 12-13】 声明并使用一个方法（代码 12-13. java）。

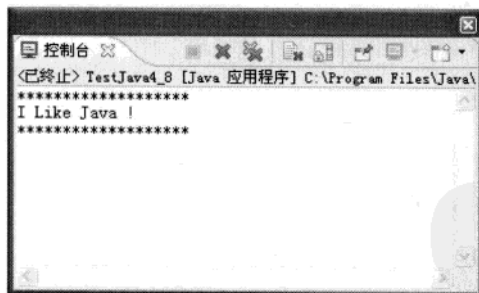
```

01 // 以下程序主要说明如何去声明并使用一个方法
02 public class TestJavashengming
03 {
04     public static void main(String args[])
05     {
06         star();           // 调用 star() 方法
07         System.out.println("I Like Java !");
08         star();           // 调用 star() 方法
09     }
10
11     public static void star()    // star() 方法
12     {
13         for(int i=0;i<19;i++)
14             System.out.print("*"); // 输出 19 个星号
15         System.out.print("\n");   // 换行
16     }
17 }

```

【运行结果】

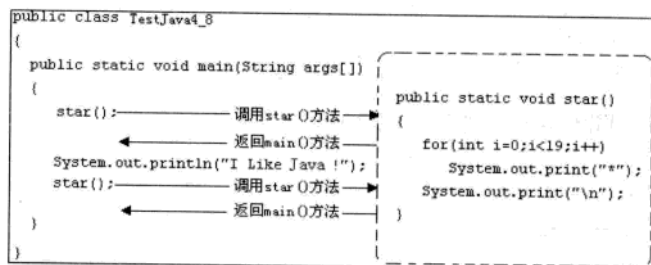
保存并运行程序，结果如图所示。



TestJava4_8 中声明了两个方法，分别为 main()方法和 star()方法。因为 main()方法是程序进入的起点，所以把调用 star()的程序代码编写在 main()里。在 main()的第 6 行调用 start() 方法，此时程序的运行流程便会进到第 11~16 行的 star()方法里执行。执行完毕，程序返回 main()方法，继续运行第 7 行，输出 “I Like Java !” 字符串。

接着在第 8 行又调用 star()方法，程序再度进到第 11~16 行的 star()方法里运行。运行完毕，返回 main()方法里，因 main()方法接下来已经没有程序代码可供执行，于是结束程序 TestJava4_8。

从本程序中可以很清楚地看出，当调用方法时，程序会跳到被调用的方法里去运行，结束后则返回原调用处继续运行。在 TestJava4_8 中，调用与运行 star()方法的流程如图所示。



不知道读者是否注意到，在程序 TestJava4_8 中 star() 方法并没有任何返回值，所以 star() 方法前面加上了一个 void 关键字。此外，因为 star() 没有传递任何的参数，所以 star() 方法的括号内保留空白即可。

至于在 star() 方法之前要加上 static 关键字，这是因为 main() 方法本身也声明成 static，而在 static 方法内只能访问到 static 成员变量（包括数据成员和方法成员）之故，因 star() 方法被 main() 方法所调用，自然也要把 star() 声明成 static 才行。如果此时还不了解 static 的真正用意也没有关系，本书将在以后的章节中对 static 关键字做详尽的介绍。

12.5.1 方法的参数与返回值

如果方法有返回值，则在声明方法之前就必须指定返回值的数据类型。相同的，如果有参数要传递到方法内，则在方法的括号内必须填上该参数及其类型。TestJava4_9 是用来说明方法的使用的另一个范例，它可以接收一个整数参数 n，输出 2*n 个星号后返回整数 2*n。

【范例 12-14】 关于方法的返回类型是整型的范例（代码 12-14.java）。

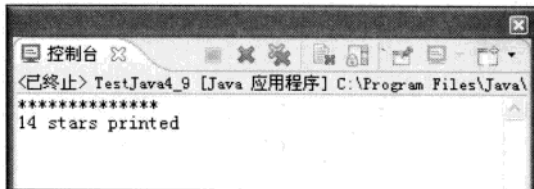
```

01 // 以下程序是关于方法的返回类型是整型的范例
02 public class TestJavafanhuihengxing
03 {
04     public static void main(String args[])
05     {
06         int num;
07         num=star(7); // 输入 7 给 star(), 并以 num 接收返回的数值
08         System.out.println(num+" stars printed");
09     }
10
11     public static int star(int n) // star() method
12     {
13         for(int i=1;i<=2*n;i++)
14             System.out.print("*"); // 输出 2*n 个星号
15         System.out.print("\n"); // 换行
16         return 2*n; // 返回整数 2*n
17     }
18 }

```


【运行结果】

保存并运行程序，结果如图所示。



在 TestJava4_9 中，因 star() 传递整数值，所以第 11 行的声明要在 star() 方法之前加上 int 关键字。此外，因要传入一个整数给 star()，所以在 star() 的括号内也要注明参数的名称与数据类型。

如果要传递一个参数，只要在方法的括号内填上所要传入的参数名称与类型即可。TestJava4_10 是一个关于计算长方形对角线长度的范例，其中 show_length() 方法用来接收长方形的宽与高，计算后返回对角线的长度。

【范例 12-15】 方法的使用 (代码 12-15.java)。

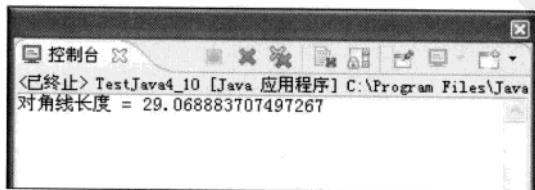
```

01 // 以下程序说明了方法的使用
02 public class TestJavafangfa
03 {
04     public static void main(String args[])
05     {
06         double num;
07         num=show_length(22,19);           // 输入 22 和 19 两个参数到 show_length()里
08         System.out.println("对角线长度 = "+num);
09     }
10
11     public static double show_length(int m, int n)
12     {
13         return Math.sqrt(m*m+n*n);       // 返回对角线长度
14     }
15 }

```

【运行结果】

保存并运行程序，结果如图所示。



TestJava4_10 的第 7 行调用 show_length(22,19)，把整数 22 和 19 传入 show_length() 方法中。

第 13 行则利用 Math 类里的 sqrt() 方法计算对角线长度。而 sqrt(n) 的作用是将参数 n 开根号。因 sqrt() 的返回值是 double 类型，因此 show_length() 的返回值也是 double 类型。

12.5.2 方法的重载

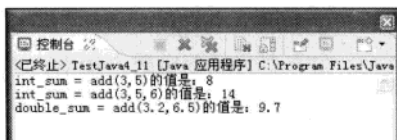
方法的重载就是在同一个类中允许同时存在一个以上的同名方法，只要它们的参数个数或类型不同即可。在这种情况下，该方法就叫被重载了，这个过程称为方法的重载。

【范例 12-16】 方法的重载（代码 12-16.java）。

```
01 // 以下程序说明了方法的重载操作
02 public class TestJavareload
03 {
04     public static void main(String[] args)
05     {
06         int int_sum ;
07         double double_sum ;
08         int_sum = add(3,5);    // 调用有两个参数的 add 方法
09         System.out.println("int_sum = add(3,5)的值是: "+int_sum);
10         int_sum = add(3,5,6); // 调用有 3 个参数的 add 方法
11         System.out.println("int_sum = add(3,5,6)的值是: "+int_sum);
12         double_sum = add(3.2,6.5); // 传入的数值为 double 类型
13         System.out.println("double_sum = add(3.2,6.5)的值是: "+double_sum);
14     }
15     public static int add(int x,int y)
16     {
17         return x+y ;
18     }
19     public static int add(int x,int y,int z)
20     {
21         return x+y+z ;
22     }
23     public static double add(double x,double y)
24     {
25         return x+y ;
26     }
27 }
```

【运行结果】

保存并运行程序，结果如图所示。



可以看到这里 add 被重载了 3 次, 但每个重载了的方法所能接受参数的个数和类型不同。相信读者现在应该明白方法重载的概念了。

12.5.3 将数组传递到方法里

方法不仅可以用来传递一般的变量, 也可用来传递数组。本小节讲述在 Java 里是如何传递数组以及如何处理方法的返回值是一维数组的问题。

1. 传递一维数组

要传递一维数组到方法里, 只要指明传入的参数是一个数组即可。TestJava4_12 是传递一维数组到 largest() 方法的一个范例, 当 largest() 接收到此数组时, 便会把数组的最大值输出。

【范例 12-17】 一维数组采用静态方式赋值 (代码 12-17. java)。

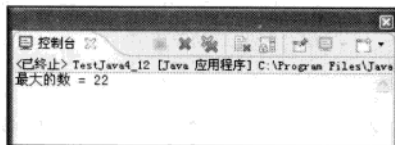
```

01 // 一维数组作为参数来传递, 这里的一维数组采用静态方式赋值
02 public class TestJavachuanshuzu
03 {
04     public static void main(String args[])
05     {
06         int score[]={7,3,8,19,6,22};           // 声明一个一维数组 score
07         largest(score);                         // 将一维数组 score 传入 largest() 方法中
08     }
09     public static void largest(int arr[])
10     {
11         int tmp=arr[0];
12         for(int i=0;i<arr.length;i++)
13             if(tmp<arr[i])
14                 tmp=arr[i];
15         System.out.println("最大的数 = "+tmp);
16     }
17 }

```

【运行结果】

保存并运行程序, 结果如图所示。



TestJava4_12 的第 11~16 行声明了 largest()方法,并将一维数组作为该方法的参数。第 9~16 行找出数组的最大值并输出。注意:如果要传递数组到方法里,只要在方法内填上数组的名称即可,如第 7 行所示。

2. 传递二维数组

二维数组的传递与一维数组类似,只要在方法里声明传入的参数是一个二维数组即可。程序 TestJava4_13 是有关传递二维数组的一个范例,把二维数组 A 传递到 print_mat()方法里,并在 print_mat()方法里输出该数组值。

【范例 12-18】 二维数组作为参数传递到方法中(代码 12-18. java)。

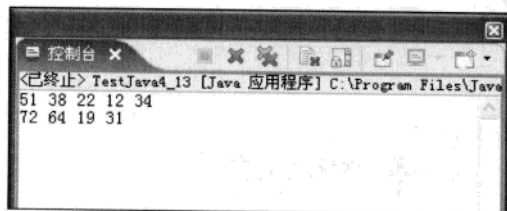
```

01 // 以下程序说明了如何将一个二维数组作为参数传递到方法中
02 public class TestJavachuanerwei
03 {
04     public static void main(String args[])
05     {
06         int A[][] = {{51,38,22,12,34},{72,64,19,31}}; // 定义一个二维数组 A
07
08         print_mat(A);
09     }
10
11     public static void print_mat(int arr[][]) // 接收整数类型的二维数组
12     {
13         for(int i=0;i<arr.length;i++)
14         {
15             for(int j=0;j<arr[i].length;j++)
16                 System.out.print(arr[i][j]+" "); // 输出数组值
17             System.out.print("\n"); // 换行
18         }
19     }
20 }

```

【运行结果】

保存并运行程序,结果如图所示。



TestJava4_13 的第 11~19 行声明了 print_mat()方法,它可以接收二维数组,并利用两个 for 循环输出数组的值。注意:可以利用.length 取出数组的行数或列数,如第 13 行和第 15 行所示。

3. 返回数组的方法

如果方法返回整数,则必须在声明时在方法的前面加上 `int` 关键字。相反,如果返回的是一维的整型数组,则必须在方法的前面加上 `int[]`。若是返回二维的整型数组,则应加上 `int[][]`,依次类推。

`TestJava4_14.java` 是返回二维数组的一个范例。将一个二维数组传入 `add10()` 方法中,在 `add10()` 方法内将每一个元素加 10 之后返回它,最后在 `main()` 里输出此数组。

【范例 12-19】 说明方法中返回一个二维数组的实现过程(代码 12-19. java)。

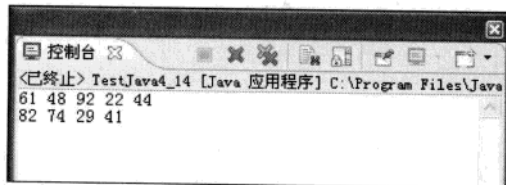
```

01 // 以下的程序说明了方法中返回一个二维数组的实现过程
02 public class TestJavafanerwei
03 {
04     public static void main(String args[])
05     {
06         int A[][]={{51,38,82,12,34},{72,64,19,31}}; // 定义二维数组
07         int B[][]=new int[2][5];
08         B=add10(A); // 调用 add10(), 并把返回的值设给数组 B
09         for(int i=0;i<B.length;i++) // 输出数组的内容
10         {
11             for(int j=0;j<B[i].length;j++)
12                 System.out.print(B[i][j]+" ");
13             System.out.print("\n");
14         }
15     }
16
17     public static int[][] add10(int arr[][])
18     {
19         for(int i=0;i<arr.length;i++)
20             for(int j=0;j<arr[i].length;j++)
21                 arr[i][j]+=10; // 将数组元素加 10
22         return arr; // 返回二维数组
23     }
24 }

```

【运行结果】

保存并运行程序,结果如图所示。



虽然 TestJava4_14 的程序代码有点长，但还是很好理解的。第 17 行赋值 add10()是可接收二维数组，且返回类型是二维的整型数组。第 21 行是完成了在循环内将数组元素值加 10 的操作，而运算之后的结果再由第 22 行的 return 语句返回。

12.6 引用数据类型的传递

在本书的开始就已经提到过，Java 中使用引用来取代 C++中的指针，那么什么是引用？Java 又是怎样通过引用来取代 C++中指针的呢？请读者先看下面程序的代码。

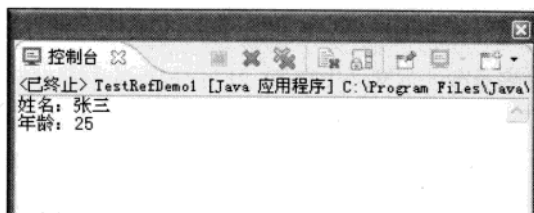
【范例 12-20】 在 Java 中引用数据类型的传递（代码 12-20.java）。

```

01  class Person
02  {
03      String name ;
04      int age ;
05  }
06  public class TestRefDemo1
07  {
08      public static void main(String[] args)
09      {
10          // 声明一个对象 p1，此对象的值为 null，表示未实例化
11          Person p1 = null ;
12          // 声明一个对象 p2，此对象的值为 null，表示未实例化
13          Person p2 = null ;
14          // 实例化 p1 对象
15          p1 = new Person() ;
16          // 为 p1 对象中的属性赋值
17          p1.name = "张三" ;
18          p1.age = 25 ;
19          // 将 p1 的引用赋给 p2
20          p2 = p1 ;
21          // 输出 p2 对象中的属性
22          System.out.println("姓名: "+p2.name);
23          System.out.println("年龄: "+p2.age);
24          p1 = null ;
25      }
26  }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~5 行声明了一个 Person 类，有 name 和 age 两个属性。

第 11、13 行分别声明了两个 Person 的对象 p1 和 p2，但这两个对象在声明时都同时赋值为 null，表示此对象未实例化。

第 15 行为对象 p1 进行实例化。

第 17、18 行分别为 p1 对象中的属性赋值。

第 20 行将 p1 的引用赋给 p2，此时相当于 p1 和 p2 都同时指向同一块堆内存。

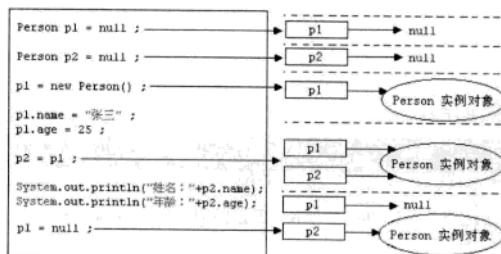
第 22、23 行分别调用 p2.name 和 p2.age 输出 p2 对象中的属性。

第 24 行把 p1 对象赋值为 null，表示此对象不再引用任何内存空间。

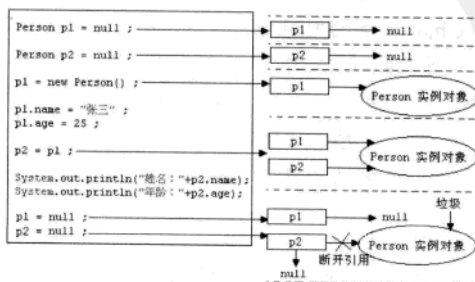
程序执行到第 24 行时，实际上 p1 断开了对其之前实例化对象的引用，而 p2 则继续指向 p1 原先的引用。

【范例分析】

从程序中可以看到，程序中并未用 new 关键字为对象 p2 实例化，而到最后依然可以用 p2.name 和 p2.age 方式输出属性的内容，而且内容与 p1 对象中的内容相似，也就是说在这个程序之中 p2 是通过 p1 对象实例化的，或者说 p1 将其自身的引用传递给了 p2。过程如图所示。



提示：如果在程序的最后又加了一段代码，令 p2=null，则之前由 p1 创建的实例化对象不再有任何对象使用它，则此对象称为垃圾对象，如图所示。



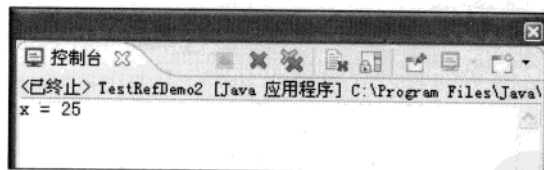
所谓垃圾对象，就是指程序中不再使用的对象引用。

【范例 12-21】 引用数据类型的传递（代码 12-21. java）。

```
01  class Change
02  {
03      int x = 0 ;
04  }
05
06  public class TestRefDemo2
07  {
08      public static void main(String[] args)
09      {
10          Change c = new Change() ;
11          c.x = 20 ;
12          fun(c) ;
13          System.out.println("x = "+c.x);
14      }
15      public static void fun(Change c1)
16      {
17          c1.x = 25 ;
18      }
19  }
```

【运行结果】

保存并运行程序，结果如图所示。



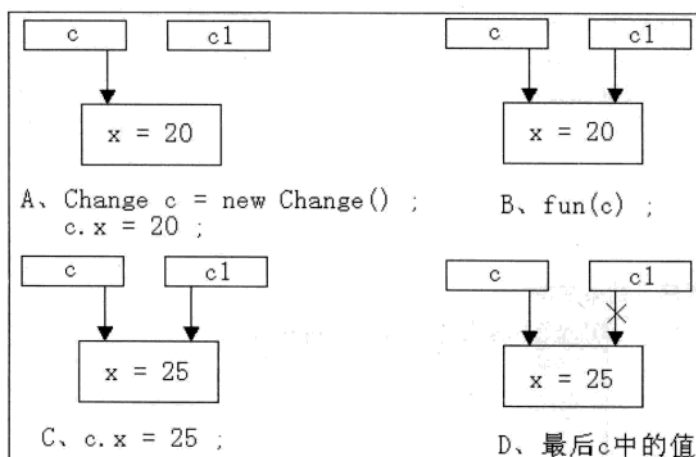
【代码详解】

第 1~4 行声明了一个名为 Change 的类，里面有一个属性 x。
第 10 行实例化了一个 Change 类的对象 c。
第 11 行将对象 c 中的 x 属性赋值为 20。
第 12 行调用 fun() 方法，将对象 c 传递到方法之中。
第 15~18 行声明这个 fun 方法，接收参数类型为 Change 类型。
第 17 行将对象 c1 中的 x 属性赋值为 25。

【范例分析】

可以看到程序最后的输出结果为“x = 25”，而程序只有在 fun() 方法中才将 x 的值赋为 25，

为什么方法完成之后值还依然被保留下来了呢？读者可以看到，在第 15 行 fun()方法接收的参数是 Change c1，也就是说 fun()方法接收的是一个对象的引用，所以在 fun 方法中所做的操作就会影响原先的参数。过程如图所示。



从图中可以看到，最开始声明的对象 `c` 将其内部的 `x` 赋值为 20 (图 A)，之后调用 `fun()` 方法，将对象赋值给 `c1`，`c1` 再继续修改 `x` 的值，此时 `c1` 与 `c` 同时指向同一个内存空间，所以 `c1` 操作了 `x` 也就相当于 `c` 操作了 `x`，因此 `fun()` 方法执行完毕，`x` 的值为 25。

12.7 覆写 Object 类中的 equals 方法

前面已经介绍过，Object 是所有类的父类，其中的 `toString()` 方法是需要被覆写的，如果读者去查 JDK 手册，会发现在 Object 类中有一个 `equals` 方法，此方法用于比较对象是否相等，而且此方法必须被覆写。为什么要覆写它呢？请看下面的范例，这是一个没有覆写 `equals()` 方法的范例。

【范例 12-22】 覆写 Object 类中的 `equals` 方法 (代码 12-22.java)。

```

01 class Person
02 {
03     private String name ;
04     private int age ;
05     public Person(String name,int age)
06     {
07         this.name = name ;
08         this.age = age ;
09     }
10 }
11 class TestOverEquals1
12 {

```

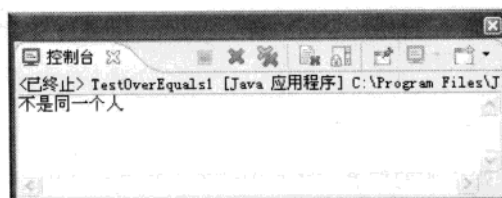
```

13     public static void main(String[] args)
14     {
15         Person p1 = new Person("张三",25);
16         Person p2 = new Person("张三",25);
17         // 判断 p1 和 p2 的内容是否相等
18         System.out.println(p1.equals(p2)?"是同一个人! ":"不是同一个人");
19     }
20 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~10 行声明了一个 Person 类，并声明一个构造方法为类的属性初始化。

第 15、16 行声明了两个 Person 对象 p1、p2，其内容相等。

第 18 行是比较两个对象的内容是否相等。

从程序中可以看到，两个对象的内容完全相等，但为什么比较的结果是不相等呢？这是因为 p1 与 p2 的内容分别在不同的内存空间，指向了不同的内存地址，所以在用 equals 比较时，实际上是调用了 Object 类中的 equals 方法。但可以看到此方法并不好用，所以在开发中往往需要覆写 equals 方法，请看下面的范例。

【范例 12-23】 equals 方法的覆写（代码 12-23. java）。

```

01     class Person
02     {
03         private String name ;
04         private int age ;
05         public Person(String name,int age)
06         {
07             this.name = name ;
08             this.age = age ;
09         }
10         // 覆写父类（Object 类）中的 equals 方法
11         public boolean equals(Object o)
12         {
13             boolean temp = true ;

```



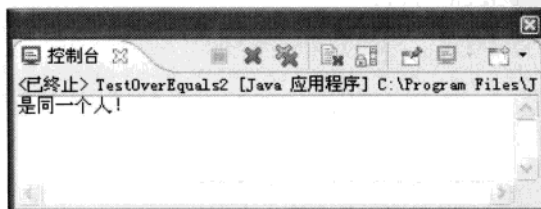
```

14      // 声明一个 p1 对象，此对象实际上就是当前调用 equals 方法的对象
15      Person p1 = this ;
16      // 判断 Object 类对象是否是 Person 的实例
17      if(o instanceof Person)
18      {
19          // 如果是 Person 类实例，则进行向下转型
20          Person p2 = (Person)o ;
21          // 调用 String 类中的 equals 方法
22          if(!(p1.name.equals(p2.name)&& p1.age==p2.age))
23          {
24              temp = false ;
25          }
26      }
27      else
28      {
29          // 如果不是 Person 类实例，则直接返回 false
30          temp = false ;
31      }
32      return temp ;
33  }
34  }
35  class TestOverEquals2
36  {
37      public static void main(String[] args)
38      {
39          Person t_p1 = new Person("张三",25);
40          Person t_p2 = new Person("张三",25);
41          // 判断 p1 和 p2 的内容是否相等
42          System.out.println(t_p1.equals(t_p2)? "是同一个人!" : "不是同一个人");
43      }
44  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~34 行声明了一个 Person 类，并在类中覆写了 Object 类的 equals() 方法。

第 15 行声明了一个 Person 对象 p1，并用 this 实例化。此时，this 就相当于当前调用此方法的对象，也就是第 42 行的 t_p1 对象。

第 17 行判断传进去的实例对象 o 是否属于 Person 类的实例化对象，如果是，则进行转型，否则返回 false。

第 22 行分别比较两个对象的内容是否相等，如果不相等，则返回 false。

第 42 行通过 t_p1 调用 equals 方法，并将 t_p2 对象的实例传到 equals 方法之中，比较两个对象是否相等。

12.8 接口对象的实例化

前面已经介绍过接口的概念，相信读者应该已经清楚了，接口是无法直接实例化的，因为接口中没有构造方法。但是却可以根据对象多态性的概念，通过接口的子类对其进行实例化，请看下面的范例。

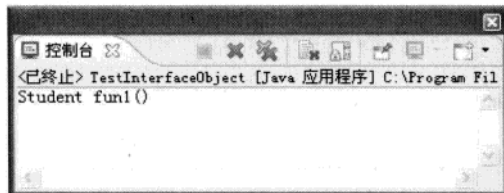
【范例 12-24】 接口对象的实例化使用实例 1（代码 12-24. java）。

```
01 interface Person
02 {
03     public void fun1();
04 }
05 class Student implements Person
06 {
07     public void fun1()
08     {
09         System.out.println("Student fun1()");
10     }
11 }
12 class TestInterfaceObject
13 {
14     public static void main(String[] args)
15     {
16         Person p = new Student();
17         p.fun1();
18     }
19 }
```

【运行结果】

保存并运行程序，结果如图所示。

200



【代码详解】

第 1~4 行声明了一个 Person 接口，此接口中只有一个抽象方法 fun1()。

第 5~11 行声明了一个 Student 类，此类实现 Person 接口，并覆写 fun1() 方法。

第 16 行声明了一个 Person 接口的对象 p，并通过其子类 Student 类去实例化此对象。

第 17 行调用 fun1() 方法，此时调用的是子类中覆写了的 fun1() 方法。

从程序中可以看到，接口是可以被实例化的，但是不能被直接实例化，只能通过其子类进行实例化。而在这里将 Person 声明为抽象类的道理也是一样的，这要留给读者自己去完成。

那么这样去实例化到底有什么好处呢？举一个例子来说明，读者应该知道现在已经有了很多的 USB 设备，无论是移动硬盘还是 MP3，这些设备都具备一个重要的特点，就是都拥有一个 USB 接口，而电脑上也有相应的插槽，所以这些设备才能正常使用。以此为例编写的程序如下。

【范例 12-25】 接口对象的实例化使用实例 2（代码 12-25.java）。

```

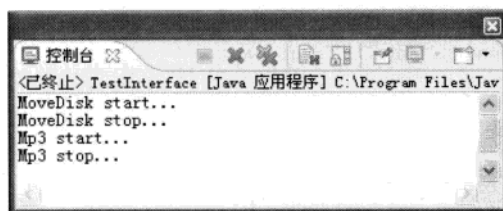
01  interface Usb
02  {
03      public void start();
04      public void stop();
05  }
06  class MoveDisk implements Usb
07  {
08      public void start()
09      {
10          System.out.println("MoveDisk start...");
11      }
12      public void stop()
13      {
14          System.out.println("MoveDisk stop...");
15      }
16  }
17  class Mp3 implements Usb
18  {
19      public void start()
20      {
21          System.out.println("Mp3 start...");
22      }

```

```
23     public void stop()
24     {
25         System.out.println("Mp3 stop...");
26     }
27 }
28
29 class Computer
30 {
31     public void work(Usb u)
32     {
33         u.start();
34         u.stop();
35     }
36 }
37
38 class TestInterface
39 {
40     public static void main(String[] args)
41     {
42         new Computer().work(new MoveDisk());
43         new Computer().work(new Mp3());
44     }
45 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~5 行声明了一个 Usb 接口，此接口中有两个抽象方法：start()、stop()。

第 6~16 行声明了一个 MoveDisk 类，此类实现 Usb 接口，并覆写了里面的两个抽象方法。

第 17~27 行声明了一个 Mp3 类，此类实现 Usb 接口，并覆写了里面的两个抽象方法。

第 29~36 行声明了一个 Computer 类，在类中有一个 work() 方法，此方法接收 Usb 对象的实例，并调用 start 和 stop 方法。

第 42、43 行分别用 Computer 类的实例化对象调用 work() 方法，并根据传进对象的不同而产生不同的结果。

从程序中可以看到，使用接口实际上就是定义出了一个统一的标准。在后面，还会介绍接口的其他使用方法。

12.9 this 关键字的使用

本节视频教学录像：39 分钟

在整个 Java 的面向对象程序设计中，this 是一个比较难理解的关键字。读者应该还记得在前面调用类内部方法时，曾提到过 this 强调对象本身，那什么是对象本身呢？其实在这里读者只要遵循一个原则即可，就是 this 表示当前对象，而所谓的当前对象就是指调用类中方法或属性的那个对象。先来看看下面的程序片段。

【范例 12-26】 在构造方法中声明与方法同名参数（代码 12-26.java）。

```
01 class Person
02 {
03     private String name ;
04     private int age ;
05     public Person(String name,int age)
06     {
07         name = name ;
08         age = age ;
09     }
10 }
```

看了上面的程序可能会有些迷惑，程序的本意是通过构造方法对 name 和 age 进行初始化的，但是在构造方法中声明的两个参数的名称也同样是 name 和 age，这就造成了一种不清楚的关系，到底第 7 行的形参 name 是赋给了类中的属性 name，还是类中的属性 name 赋给了形参中的 name？为了避免出现这种混淆，可以采用 this 这种方式，请看修改后的代码。

【范例 12-27】 this 关键字的使用范例 1（代码 12-27.java）。

```
01 class Person-1
02 {
03     private String name ;
04     private int age ;
05     public Person(String name,int age)
06     {
07         this.name = name ;
08         this.age = age ;
09     }
10 }
```

Person-1.java 这段代码与 Person.java 的不同之处在于在第 7、8 行分别加上了 this 关键字。

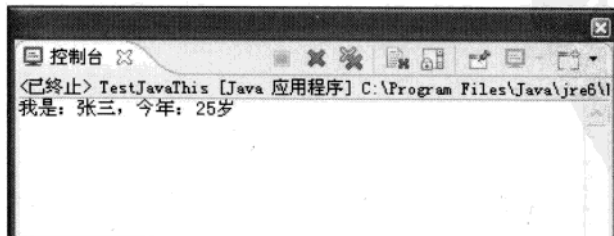
还记得之前说过的 this 表示当前对象吗？那么此时的 this.name 和 this.age 就分别代表类中的 name 和 age 属性，这个时候再完成赋值操作的话，就可以清楚地知道谁赋值给谁了。完整程序代码如下。

【范例 12-28】 this 关键字的使用范例 2（代码 12-28.java）。

```
01 class Person
02 {
03     private String name ;
04     private int age ;
05     public Person(String name,int age)
06     {
07         this.name = name ;
08         this.age = age ;
09     }
10     public String talk()
11     {
12         return "我是："+name+"，今年："+age+"岁"；
13     }
14 }
15 public class TestJavaThis
16 {
17     public static void main(String[] args)
18     {
19         Person p = new Person("张三",25)；
20         System.out.println(p.talk())；
21     }
22 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~14 行声明了一个名为 Person 的类。

第 5~9 行声明了 Person 类的一个构造方法，此构造方法的作用是对类中的属性赋初值。

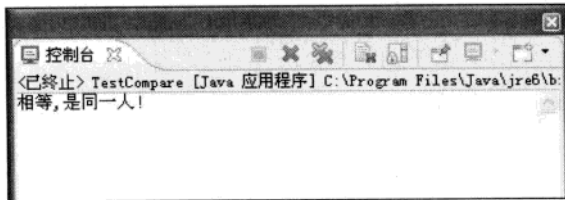
为了更好地说明 this 是表示当前对象的，下面再举一个范例：判断两个对象是否相等，在这里假设只要姓名、年龄都相同的就为同一个人，否则就不是同一个人。

【范例 12-29】 判断两个对象是否相等（代码 12-29. java）。

```
01  class Person
02  {
03      String name ;
04      int age ;
05      Person(String name,int age)
06      {
07          this.name = name ;
08          this.age = age ;
09      }
10      boolean compare(Person p)
11      {
12          if(this.name.equals(p.name)&&this.age==p.age)
13          {
14              return true ;
15          }
16          else
17          {
18              return false ;
19          }
20      }
21  }
22
23  public class TestCompare
24  {
25      public static void main(String[] args)
26      {
27          Person p1 = new Person("张三",30);
28          Person p2 = new Person("张三",30);
29          System.out.println(p1.compare(p2)?"相等,是同一人!":"不相等,不是同一人!");
30      }
31  }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~21 行声明了一个名为 Person 的类，里面有一个构造方法和一个比较方法。

第 10~20 行在 Person 类中声明了一个 compare() 方法，此方法接收 Person 实例对象的引用。

第 12 行比较姓名和年龄是否同时相等。

第 29 行由 p1 调用 compare() 方法，将 p2 传入到 compare 方法之中，所以第 12 行的 this.name 就代表 p1.name，this.age 就代表 p1.age，而传入的参数 p2 则被用 compare() 方法中的参数 p 表示。

【范例分析】

由此不难理解 this 是表示当前对象这一重要概念，所以程序的最后输出了“相等，是同一人！”的正确判断信息。

如果在程序中想用某一个构造方法调用另一个构造方法，也可以用 this 来实现。具体的调用形式如下。

```
this();
```

【范例 12-30】 用 this 调用构造方法（代码 12-30.java）。

```
01  class Person
02  {
03      String name ;
04      int age ;
05      public Person()
06      {
07          System.out.println("1. public Person()");
08      }
09      public Person(String name,int age)
10      {
11          // 调用本类中无参构造方法
12          this();
13          this.name = name ;
14          this.age = age ;
15          System.out.println("2. public Person(String name,int age)");
16      }
17  }
18  public class TestJavaThis1
```

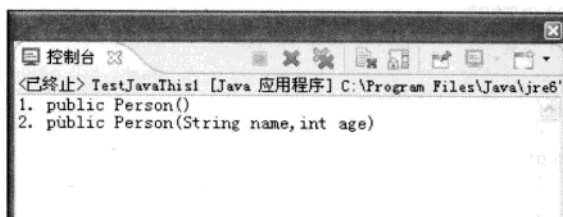
```

19  {
20      public static void main(String[] args)
21      {
22          new Person("张三",25);
23      }
24  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~17 行声明了一个名为 Person 的类，类中声明了一个无参、一个有参的构造方法。

第 12 行使用 this()调用本类中的无参构造方法。

第 22 行声明一个 Person 类的匿名对象，调用了有参的构造方法。

【范例分析】

从程序 TestJavaThis1.java 可以看到，在第 22 行虽然调用了 Person 中有两个参数的构造方法，但由于第 12 行使用了 this()调用本类中的无参构造方法，所以程序先去执行 Person 中的无参构造方法，之后再去继续执行其他的构造方法。



提示：有的读者经常会有这样的疑问：如果我把 this()调用无参构造方法的位置任意调换，那不就可以在任何时候都可以调用构造方法了么？实际上这样理解是错误的。构造方法是在实例化一个对象时被自动调用的，也就是说在类中的所有方法里，只有构造方法是被优先调用的，所以使用 this 调用构造方法必须也只能放在类中。

12.10 static 关键字的使用

本节视频教学录像：46 分钟

在 Java 中，可以使用 static 关键字声明静态变量和方法。

12.10.1 静态变量

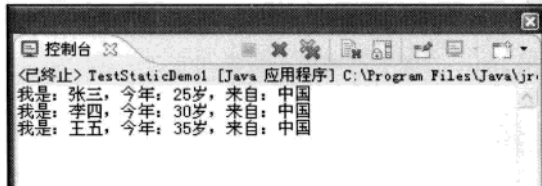
在程序中如果用 static 声明变量的话，则此变量称为静态变量。那什么是静态变量？使用静态变量又有什么好处呢？读者先来看看下面的范例。

【范例 12-31】 静态变量的声明（代码 12-31. java）。

```
01 class Person
02 {
03     String name ;
04     String city ;
05     int age ;
06     public Person(String name, String city, int age)
07     {
08         this.name = name ;
09         this.city = city ;
10         this.age = age ;
11     }
12     public String talk()
13     {
14         return "我是: "+this.name+", 今年: "+this.age+"岁, 来自: "+this.city;
15     }
16 }
17 public class TestStaticDemo1
18 {
19     public static void main(String[] args)
20     {
21         Person p1 = new Person("张三","中国",25);
22         Person p2 = new Person("李四","中国",30);
23         Person p3 = new Person("王五","中国",35);
24         System.out.println(p1.talk());
25         System.out.println(p2.talk());
26         System.out.println(p3.talk());
27     }
28 }
```

【运行结果】

保存并运行程序，结果如图所示。

**【代码详解】**

第 1~16 行声明了一个名为 Person 的类，含有 3 个属性：name、age、city。

第 6~11 行声明了 Person 类的一个构造方法，此构造方法分别对各属性赋值。

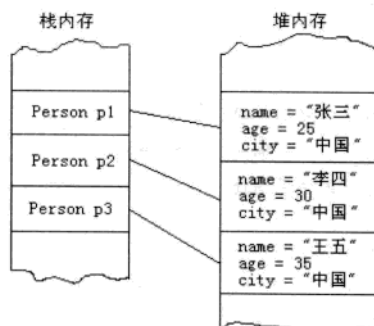
第 12~15 行声明了一个 talk() 方法，此方法用于返回用户信息。

第 21~23 行分别实例化 3 个 Person 对象。

第 24~26 行分别调用类中的 talk() 方法输出用户信息。

【范例分析】

从程序中可以看到，所有的 Person 对象都有一个 city 属性，而且所有的属性也全部相同，如图所示。



读者可以想一想，假设程序产生了 50 个 Person 对象，如果想修改所有人的 city 属性，是不是就要调用 50 遍 city 属性重新修改，这显然太麻烦了。所以在 Java 中提供了 static 关键字，用它来修饰类的属性后，则此属性就是公共属性了。将 TestStaticDemo1.java 程序稍作修改就形成了范例 TestStaticDemo2.java，如下所示。

【范例 12-32】 static 关键字的使用（代码 12-32.java）。

```

01  class Person
02  {
03      String name ;
04      static String city = "中国";
05      int age ;
06      public Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;
10      }
11      public String talk()
12      {
13          return "我是: "+this.name+"，今年: "+this.age+"岁，来自: "+city;
14      }
15  }
16  public class TestStaticDemo2
17  {
18      public static void main(String[] args)

```

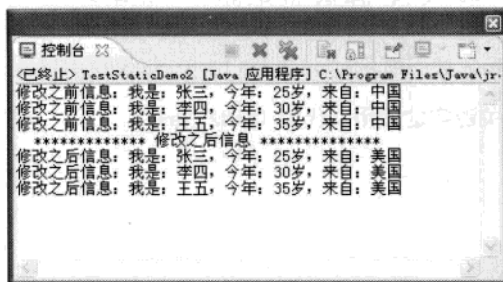
```

19     {
20         Person p1 = new Person("张三",25);
21         Person p2 = new Person("李四",30);
22         Person p3 = new Person("王五",35);
23         System.out.println("修改之前信息: "+p1.talk());
24         System.out.println("修改之前信息: "+p2.talk());
25         System.out.println("修改之前信息: "+p3.talk());
26         System.out.println(" ***** 修改之后信息 *****");
27         // 修改后的信息
28         p1.city = "美国";
29         System.out.println("修改之后信息: "+p1.talk());
30         System.out.println("修改之后信息: "+p2.talk());
31         System.out.println("修改之后信息: "+p3.talk());
32     }
33 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~15 行声明了一个名为 Person 的类，含有 3 个属性：name、age、city。其中 city 为 static 类型。

第 6~10 行声明了 Person 类的一个构造方法，此构造方法的作用是分别对 name、age 属性赋值。

第 11~14 行声明了一个 talk() 方法，此方法用于返回用户信息。

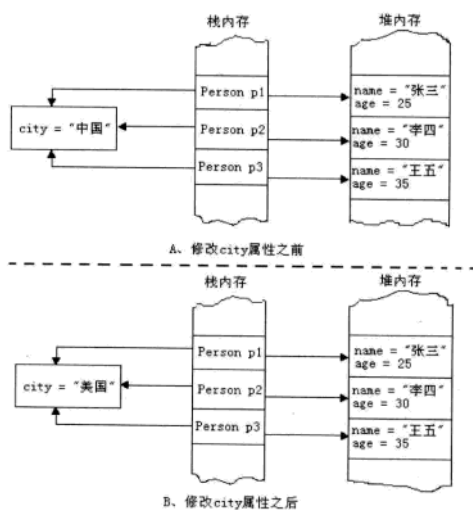
第 20~22 行分别实例化 3 个 Person 对象。

第 23~25 行分别调用类中的 talk() 方法输出用户信息。

第 28 行修改 p1 中的 city 属性。

【范例分析】

从程序中可以看到，只在第 28 行修改了 city 属性，而且只修改了一个对象的 city 属性，但再次输出时，可以看到全部对象的 city 值都发生了同样的变化，这说明用 static 声明的属性是所有对象共享的。如图所示。



从图中可以看到，所有的对象都指向同一个 city 属性，只要当中有一个对象修改了 city 属性的内容，则所有的对象都会被同时修改。

另外读者也需要注意一点：用 static 方式声明的属性，也可以用类名直接访问。拿上面的程序来说，如果想修改 city 属性中的内容，可以用如下的方式。

```
Person.city = "美国";
```

所以有些书上也把用 static 类型声明的变量称之为“类变量”。



提示：既然 static 类型的变量是所有对象共享的内存空间，也就是说无论最终有多少个对象产生，也都只有一个 static 类型的属性，那可不可以用它来计算类到底产生了多少个实例对象呢？读者可以想一想，只要一个类产生一个新的实例对象，就都会去调用构造方法，所以可以在构造方法中加入一些记数操作。

12.10.2 静态方法

static 既可以在声明变量时使用，也可以用其来声明方法，用它声明的方法有时也被称为“类方法”。请看下面的范例。

【范例 12-33】 静态方法的声明（代码 12-33. java）。

```
01 class Person
02 {
03     String name ;
04     private static String city = "中国";
05     int age ;
06     public Person(String name,int age)
07     {
```

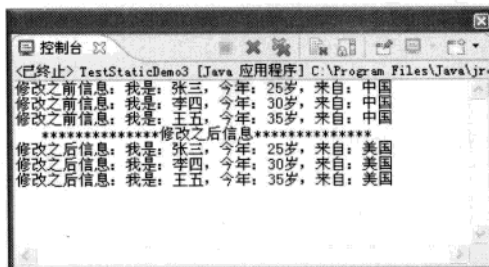
```

08         this.name = name ;
09         this.age = age ;
10     }
11     public String talk()
12     {
13         return "我是: "+this.name+", 今年: "+this.age+"岁, 来自: "+city;
14     }
15     public static void setCity(String c)
16     {
17         city = c ;
18     }
19 }
20 public class TestStaticDemo3
21 {
22     public static void main(String[] args)
23     {
24         Person p1 = new Person("张三",25);
25         Person p2 = new Person("李四",30);
26         Person p3 = new Person("王五",35);
27         System.out.println("修改之前信息: "+p1.talk());
28         System.out.println("修改之前信息: "+p2.talk());
29         System.out.println("修改之前信息: "+p3.talk());
30         System.out.println(" *****修改之后信息*****");
31         // 修改后的信息
32         Person.setCity("美国");
33         System.out.println("修改之后信息: "+p1.talk());
34         System.out.println("修改之后信息: "+p2.talk());
35         System.out.println("修改之后信息: "+p3.talk());
36     }
37 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~19 行声明了一个名为 Person 的类，类中含有一个 static 类型的变量 city，并对此对象进行了封装。

第 5~18 行声明了一个 static 类型的方法，此方法也可以用类名直接调用，用于修改 city 属性的内容。

第 32 行由 Person 调用 setCity() 方法，对 city 的内容进行修改。



提示：在使用 static 类型声明的方法时需要注意的是：如果在类中声明了一个 static 类型的属性，则此属性既可以在非 static 类型的方法中使用，也可以在 static 类型的方法中使用。但若要用 static 类型的方法调用非 static 类型的属性，就会出现错误。

12.10.3 理解 main() 方法

在前面的章节中已经多次提到，如果一个类要被 Java 解释器直接装载运行，这个类中必须有 main() 方法。有了前面所学的知识，读者现在可以理解 main() 方法的含义了。

由于 Java 虚拟机需要调用类的 main() 方法，所以该方法的访问权限必须是 public，又因为 Java 虚拟机在执行 main() 方法时不必创建对象，所以该方法必须是 static 的，该方法接收一个 String 类型的数组参数，该数组中保存执行 Java 命令时传递给所运行的类的参数。

向 Java 中传递参数可以使用如下的命令。

```
java 类名称 参数 1 参数 2 参数 3
```

可通过运行程序 TestMain.java 来了解如何向类中传递参数，以及程序是如何取得这些参数的。

【范例 12-34】 向类中传递参数（代码 12-34.java）。

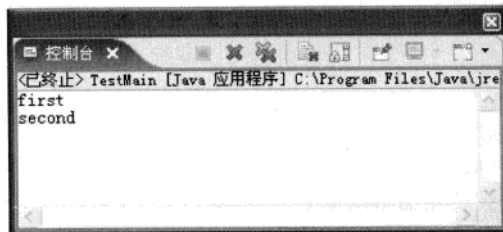
```
01 public class TestMain
02 {
03     /*
04         public: 表示公共方法
05         static: 表示此方法为一静态方法，可以由类名直接调用
06         void: 表示此方法无返回值
07         main: 系统定义的方法名称
08         String args[]: 接收运行时参数
09     */
10     public static void main(String[] args)
11     {
12         // 取得输入参数的长度
13         int j = args.length;
```



```
14      if(j!=2)
15      {
16          System.out.println("输入参数个数有错误!");
17          // 退出程序
18          System.exit(1);
19      }
20      for (int i=0;i<args.length;i++)
21      {
22          System.out.println(args[i]);
23      }
24  }
25  }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码注解】

第 14 行判断输入参数的个数是否为两个参数，如果不是，则退出程序。
所有接收的参数都被存放在 args[] 字符串数组之中，用 for 循环输出全部内容。

12.10.4 静态代码块

一个类可以使用不包含在任何方法体中的静态代码块，当类被载入时，静态代码块被执行，且只执行一次。静态代码块经常用来进行类属性的初始化。

【范例 12-35】 静态代码块的使用（代码 12-35. java）。

```
01  class Person
02  {
03      public Person()
04      {
05          System.out.println("1.public Person()");
06      }
07      // 此段代码会首先被执行
08      static
```

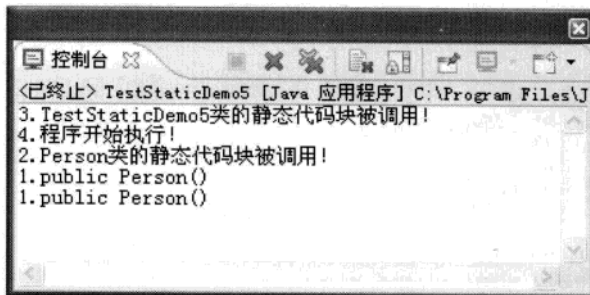
```

09      {
10          System.out.println("2.Person 类的静态代码块被调用! ");
11      }
12  }
13  public class TestStaticDemo5 {
14      // 运行本程序时, 静态代码块会被自动执行
15      static
16      {
17          System.out.println("3.TestStaticDemo5 类的静态代码块被调用! ");
18      }
19      public static void main(String[] args) {
20          System.out.println("4.程序开始执行! ");
21          // 产生两个实例化对象
22          new Person();
23          new Person();
24      }
25  }

```

【运行结果】

保存并运行程序, 结果如图所示。



【代码详解】

第 1~12 行声明了一个名为 Person 的类。

第 8~11 行声明了一个静态代码块, 此代码块放在 Person 类之中。

第 15~18 行在类 TestStaticDemo5 中也声明了一个静态代码块。

第 22、23 行分别产生了两个 Person 类的匿名对象。

【范例分析】

从程序的运行结果中可以看到, 放在 TestStaticDemo5 类中的静态代码块首先被调用, 这是因为程序首先执行 TestStaticDemo5 类, 所以此程序的静态代码块会首先被执行。程序在第 22、23 行产生了两个匿名对象, 可以看到 Person 类中的静态代码块只执行了一次, 而且静态代码块优先于静态方法, 由此得知: 静态代码块可以对静态属性初始化。

12.11 final 关键字的使用

本节视频教学录像：6 分钟

在 Java 中声明类、属性和方法时，可使用关键字 final 来修饰。

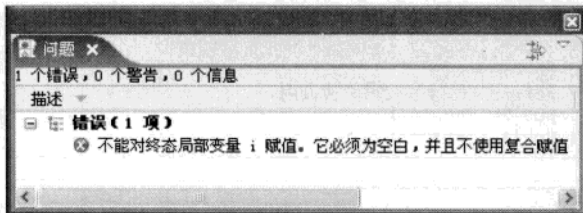
- (1) final 标记的类不能被继承。
- (2) final 标记的方法不能被子类覆写。
- (3) final 标记的变量（成员变量或局部变量）即为常量，只能赋值一次。

【范例 12-36】 final 标记的变量只能赋值一次实例（代码 12-36.java）。

```
01 class TestFinalDemo1
02 {
03     public static void main(String[] args)
04     {
05         final int i = 10 ;
06         // 修改用 final 修饰的变量 i
07         i++ ;
08     }
09 }
```

【运行结果】

保存并运行程序，结果如图所示。

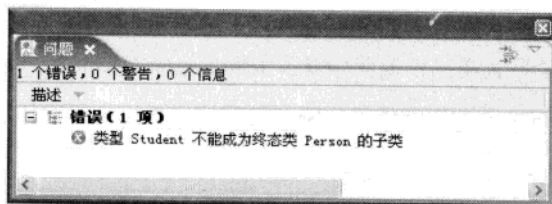


【范例 12-37】 final 标记的类不能被继承实例（代码 12-37.java）。

```
01 final class Person
02 {
03 }
04 class Student extends Person
05 {
06 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例 12-38】 final 标记的方法不能被子类覆写实例（代码 12-38. java）。

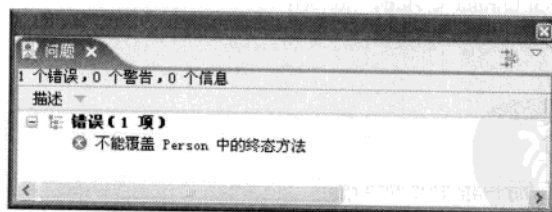
```

01  class Person
02  {
03      // 此方法声明为 final 不能被子类覆写
04      final public String talk()
05      {
06          return "Person: talk()";
07      }
08  }
09
10  class Student extends Person
11  {
12      public String talk()
13      {
14          return "Student: talk()";
15      }
16  }

```

【运行结果】

保存并运行程序，结果如图所示。



12.12 instanceof 关键字的使用

可以用 instanceof 判断一个类是否实现了某个接口，也可以用它来判断一个实例对象是否属于一个类。instanceof 的语法格式如下。

对象 instanceof 类(或接口)

它的返回值是布尔型的，或真(true)或假(false)。

将上面范例的程序进行简单的修改就构成了范例 TestJavaDemo3.java，如下所示。

【范例 12-39】 instanceof 关键字使用实例（代码 12-39.java）。

```
01  class Person
02  {
03      public void fun1()
04      {
05          System.out.println("1.Person{fun1()}");
06      }
07      public void fun2()
08      {
09          System.out.println("2.Person{fun2()}");
10      }
11  }
12
13      // Student 类继承自 Person 类，也就继承了 Person 类中的 fun1()、fun2()方法
14  class Student extends Person
15  {
16      // 在这里覆写了 Person 类中的 fun1()方法
17      public void fun1()
18      {
19          System.out.println("3.Student{fun1()}");
20      }
21      public void fun3()
22      {
23          System.out.println("4.Studen{fun3()}");
24      }
25  }
26  class TestJavaDemo3
27  {
28      public static void main(String[] args)
29      {
30          // 声明一个父类对象并通过子类对象对其进行实例化
31          Person p = new Student();
32          // 判断对象 p 是否是 Student 类的实例
33          if(p instanceof Student)
34          {
35              // 将 Person 类的对象 p 转为 Student 类型
36              Student s = (Student)p;
```



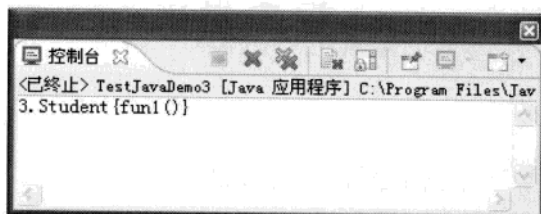
```

37         s.fun1();
38     }
39     else
40     {
41         p.fun2();
42     }
43 }
44 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码注解】

第 31 行声明了一个父类对象 p，并通过其子类实例化此对象。

第 33 行用 instanceof 关键字判断 p 对象是否是 Student 的实例。在此范例中，因为 p 是通过 Student 类实例化的，所以此条件满足。第 36 行将 p 对象强制转换为 Student 类的对象，并调用 fun1() 方法，调用此方法时，实际上调用的是被子类覆写了的 fun1() 方法。

12.13 练一练

一、填空题

1. 在 Java 中，所有的类都是由_____类衍生出来的。
2. 方法的重载是指_____。
3. 使用 static 关键字声明的变量称为_____变量。

二、简答题

简述 final 关键字标记的特性。

12.14 跟我上机

定义一个包含 “name”、“age” 和 “sex” 的对象，使用匿名对象输出对象实例。

第 13 章

储存类的仓库——Java常用类库



本章视频教学录像：5 小时 6 分钟

Java类库就是Java API(应用程序接口)，是系统提供的已实现的标准类的集合，使用Java类库可以完成涉及字符串处理、图形、网络等多方面的操作。本章将讲解API的相关概念、基本数据类型和包装类、处理映射的方法、几种类和接口的使用，以及对象克隆的相关知识。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握 Java 类库的相关概念
- ☐ 熟悉 System 类和 Runtime 类
- ☐ 熟悉 Math 和 Random 类
- ☐ 了解对象克隆相关知识



13.1 API 概念

API (Application Programming Interface)就是应用程序编程接口。

假设现在要编写一个机器人程序,去控制一个机器人踢足球,程序需要向机器人发出向前跑、向后转、射门、拦截等命令,没有编过程序的人很难想象如何编写这样的程序。但对于有编程经验的人来说,就知道机器人厂商一定会提供一些控制这些机器人的 Java 类,该类中就有操纵机器人的各种动作的方法,只需要为每个机器人安排一个该类的实例对象,再调用这个对象的各种方法,机器人就会去执行各种动作。这个 Java 类就是机器人厂家提供的应用程序编程的接口,厂家就可以对这些 Java 类美其名曰:xxx Robot API (也就是 xxx 厂家的机器人 API)。好的机器人厂家不仅会提供 Java 程序用的 Robot API,也会提供 Windows 编程语言(如 VC++)用的 Robot API,以满足各类编程人员的需要。

13.2 String 类和 StringBuffer 类

▶ 本节视频教学录像: 1 小时 49 分钟

一个字符串就是一连串的字符,字符串的处理在许多程序中都用得到。Java 定义了 String 和 StringBuffer 两个类来封装对字符串的各种操作。它们都被放到了 java.lang 包中,不需要用 import java.lang 这个语句导入该包就可以直接使用它们。

String 类用于比较两个字符串,查找和抽取串中的字符或子串,进行字符串与其他类型之间的相互转换等。String 类对象的内容一旦被初始化就不能再改变。

StringBuffer 类用于内容可以改变的字符串,可以将其他各种类型的数据增加、插入到字符串中,也可以转置字符串中原来的内容。一旦通过 StringBuffer 生成了最终想要的字符串,就应该使用 StringBuffer.toString()方法将其转换成 String 类,随后,就可以使用 String 类的各种方法操纵这个字符串了。

Java 为字符串提供了特别的连接操作符(+),可以把其他各种类型的数据转换成字符串,并前后连接成新的字符串。连接操作符(+)的功能是通过 StringBuffer 类和它的 append 方法实现的。例如:

```
String x = "a" + 4 + "c";
```

编译时等效于:

```
String x=new StringBuffer().append("a").append(4).append("c").toString();
```

在实际开发中,如果需要频繁改变字符串的内容,就需要考虑用 StringBuffer 类实现,因为其内容可以改变,所以执行性能会比 String 类更高。

13.3 基本数据类型的包装类

本节视频教学录像：39 分钟

Java 对数据既提供基本数据的简单类型，也提供了相应的包装类。使用基本数据类型，可以改善系统的性能，也能够满足大多数的应用需求。但基本数据类型不具有对象的特性，不能满足某些特殊的需求。从 JDK 中可以知道，Java 中的很多类的很多方法的参数类型都是 Object，即这些方法接收的参数都是对象，同时，又需要用这些方法来处理基本数据类型的数据，这时就要用到包装类。比如，用 Integer 类来包装整数。

读者从前面的章节中应该已经了解到 Java 中的基本数据类型共有 8 种，那么与之相对应的包装类也共有 8 种，表中列出了其对应关系。

基本数据类型	基本数据类型包装类
int	Integer
char	Character
float	Float
double	Double
byte	Byte
long	Long
short	Short
boolean	Boolean

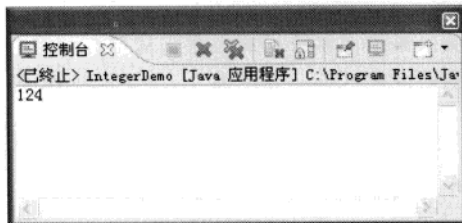
下面举一个具体的例子，告诉读者如何去使用这些包装类。

【范例 13-1】 使用包装类（代码 13-1. java）。

```
01 class IntegerDemo
02 {
03     public static void main(String[] args)
04     {
05         String a = "123";
06         int i = Integer.parseInt(a);
07         i++;
08         System.out.println(i);
09     }
10 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

本程序使用 Integer 类中的 parseInt()方法，将一个字符串转换成基本数据类型。

13.4 System 类与 Runtime 类

▶ 本节视频教学录像：27 分钟

本节介绍 Java 中两个重要的类：System 类和 Runtime 类。

13.4.1 System 类

Java 不支持全局方法和变量，Java 设计者将一些系统相关的重要方法和变量收集到了一个统一的类中，这就是 System 类。System 类中的所有成员都是静态的，而要引用这些变量和方法，可直接使用 System 类名作为前缀。在前面已经使用到了标准输入和输出的 in 和 out 变量，下面再介绍 System 类中的几个方法，其他的方法读者可以参阅 JDK 文档资料。

exit(int status)方法，提前终止虚拟机的运行。对于发生了异常情况而想终止虚拟机的运行，传递一个非零值作为参数。若在用户正常操作下终止虚拟机的运行，则传递零值作为参数。

currentTimeMillis 方法，返回自 1970 年 1 月 1 日 0 点 0 分 0 秒起至今的以毫秒为单位的时间，这是一个 long 类型的大数值。在计算机内部只有数值，没有真正的日期类型及其他各种类型，也就是说，平常用到的日期实质上就是一个数值，但通过这个数值，能够推算出其对应的具体日期时间。

getProperties 方法是获得当前虚拟机的环境属性。每一个属性都是变量与值以成对的形式出现的。

同样的道理，Java 作为一个虚拟的操作系统，它也有自己的环境属性。Properties 是 Hashtable 的子类，正好可以用于存储环境属性中的多个“变量/值”对格式的数据，getProperties 方法返回值是包含了当前虚拟机的所有环境属性的 Properties 类型的对象。下面的例子演示打印出当前虚拟机的所有环境属性的变量和值。

【范例 13-2】 打印当前虚拟机的所有环境属性的变量和值（代码 13-2. java）。

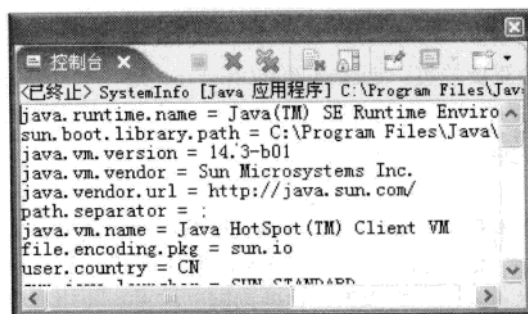
```
01 import java.util.*;
02 public class SystemInfo
03 {
04     public static void main(String[] args)
05     {
06         Properties sp=System.getProperties();
07         Enumeration e=sp.propertyNames();
08         while(e.hasMoreElements())
09         {
10             String key=(String)e.nextElement();
11             System.out.println(key+" = "+sp.getProperty(key));
12         }
13     }
14 }
```



```
13     }
14 }
```

【运行结果】

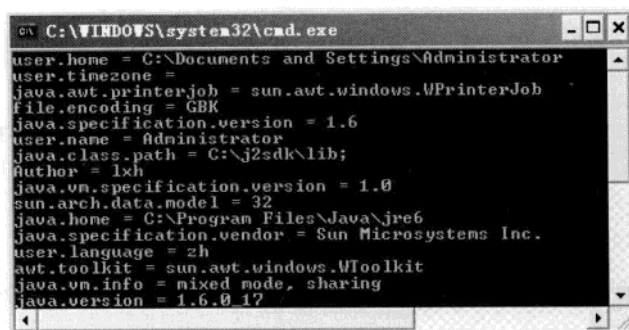
保存并运行程序，结果如图所示。



在 Windows 中增加一个新的环境属性是很容易的，但如何为 Java 虚拟机增加一个新的环境属性呢？在命令行窗口直接运行 Java 命令，在显示的用法帮助中，会看到 Java 命令有一个 -D<name>=<value>格式的选项可以设置新的系统环境属性。按下面的格式运行。

```
java -DAuthor=lxh SystemInfo
```

程序运行的结果如图所示。



可以看到输出结果中多了一行“Author = lxh”，即 Java 虚拟机中多了一个新的环境属性 Author。



提示：-D 与 Author 之间没有空格。讲解了 `getProperties` 方法后，读者也应该明白 `setProperties` 方法了。

13.4.2 Runtime 类

Runtime 类封装了 Java 命令本身的运行进程，其中的许多方法与 System 中的方法重复。不能直接创建 Runtime 实例，但可以通过静态方法 `Runtime.getRuntime` 获得正在运行的 Runtime 对象的引用。

Java 命令运行后，本身是多任务操作系统上的一个进程，在这个进程中启动一个新的进程，即执行其他程序时使用 `exec` 方法。`exec` 方法返回一个代表子进程的 `Process` 类对象，通过这个对象，Java 进程可以与子进程交互。

【范例 13-3】 Java 进程与子进程交互（代码 13-3. java）。

```
01 public class RuntimeDemo
02 {
03     public static void main(String[] args)
04     {
05         Runtime run = Runtime.getRuntime();
06         try
07         {
08             run.exec("notepad.exe");
09         }
10         catch (Exception e)
11         {
12             e.printStackTrace();
13         }
14     }
15 }
```

【范例分析】

运行程序之后，可以看到程序已经为读者打开了记事本程序。所以通过 `Runtime` 类可以为开发者执行操作系统的可执行程序。

13.5 Date 与 Calendar、DateFormat 类

▶ 本节视频教学录像：1 小时

`Date` 类用于表示日期和时间，最简单的构造方法是 `Date()`，它以当前的日期和时间初始化一个 `Date` 对象。由于开始设计 `Date` 时没有考虑到国际化，所以后来又设计了两个新的类来解决 `Date` 类中的问题，一个是 `Calendar` 类，另一个是 `DateFormat` 类。

`Calendar` 类是一个抽象基类，主要完成日期字段之间相互操作的功能。如 `Calendar.add` 方法用于实现在某一日期的基础上增加若干天（或年、月、小时、分、秒等日期字段）后的新日期，`Calendar.get` 方法用于取出日期对象中的年、月、日、小时、分、秒等日期字段的值，`Calendar.set` 方法用于修改日期对象中的年、月、日、小时、分、秒等日期字段的值。`Calendar.getInstance` 方法用于返回一个 `Calendar` 类型（更确切地说是它的某个子类）的对象实例，`GregorianCalendar` 类是 JDK 目前提供的一个唯一的 `Calendar` 子类，`Calendar.getInstance` 方法返回的就是预设了当前时间的 `GregorianCalendar` 类对象。

下面的例子计算出距当前日期时间 230 天后的日期时间，并用“xxxx 年 xx 月 xx 日 xx 小时：xx 分：xx 秒”的格式输出。

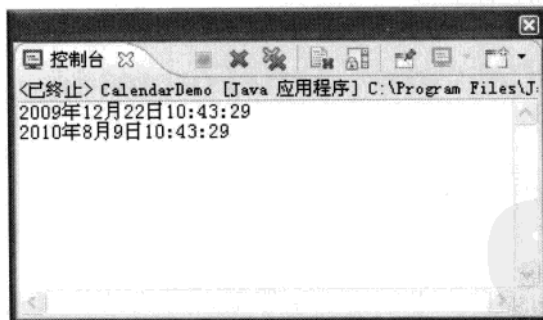
【范例 13-4】 计算出距当前日期时间 230 天后的日期时间（代码 13-4. java）。

```

01  import java.util.*;
02  public class CalendarDemo
03  {
04      public static void main(String[] args)
05      {
06          Calendar c1=Calendar.getInstance();
07          // 下面打印当前时间
08          System.out.println(c1.get(c1.YEAR)+"年"+(c1.get(c1.MONTH)+1)+
09              "月"+c1.get(c1.DAY_OF_MONTH)+"日"+c1.get(c1.HOUR)+
10              ":"+c1.get(c1.MINUTE)+":"+c1.get(c1.SECOND));
11          // 增加天数为 230
12          c1.add(c1.DAY_OF_YEAR,230);
13
14          // 下面打印的是 230 天后的时间
15          System.out.println(c1.get(c1.YEAR)+"年"+(c1.get(c1.MONTH)+1)+
16              "月"+c1.get(c1.DAY_OF_MONTH)+"日"+c1.get(c1.HOUR)+
17              ":"+c1.get(c1.MINUTE)+":"+c1.get(c1.SECOND));
18      }
19  }
    
```

【运行结果】

保存并运行程序，结果如图所示。



虽然 Calendar 类几乎完全替代了 Date 类，但在某些情况下，开发者仍有可能要用到 Date 类，譬如，程序中用的另外一个类的方法要求一个 Date 类型的参数。有时，要将用 Date 对象表示的日期以指定的格式输出，或是将用特定格式显示的日期字符串转换成一个 Date 对象，而 Java.text.DateFormat 就是实现这种功能的抽象基类。java.text.SimpleDateFormat 类是 JDK 目前提供的的一个 DateFormat 子类，它是一个具体类，具有把 Date 对象格式化为本地字符串，或者通过语义分析把日期或时间字符串转换为 Date 对象的功能。

下面的范例将“2005-8-11 18:30:38”格式的日期字符串转换成“2005 年 08 月 11 日 06 点 30 分 38 秒”的形式。

【范例 13-5】 更改日期字符串（代码 13-5. java）。

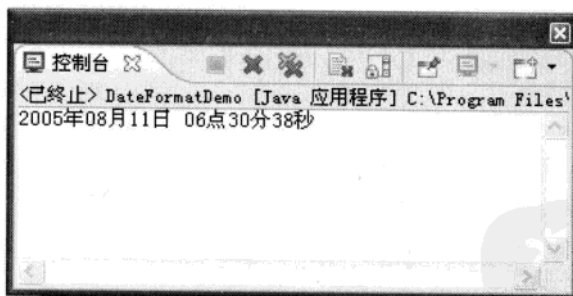
```

01  import java.text.* ;
02  import java.util.Date;
03  public class DateFormatDemo
04  {
05      public static void main(String[] args)
06      {
07          SimpleDateFormat sp1 = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
08          SimpleDateFormat sp2 = new SimpleDateFormat("yyyy 年 MM 月 dd 日 hh 点 mm 分 ss 秒");
09          try
10          {
11              Date d = sp1.parse("2005-8-11 18:30:38");
12              System.out.println(sp2.format(d));
13          }
14          catch (ParseException e)
15          {
16              e.printStackTrace();
17          }
18      }
19  }

```

【运行结果】

保存并运行程序，结果如图所示。

**【范例分析】**

SimpleDateFormat 类就相当于一个模板，其中 yyyy 对应的是年，MM 对应的是月，dd 对应的是日。更详细的细节可查阅 JDK 文档，关于这些参数，JDK 中写得非常清楚。

在程序中定义了一个 SimpleDateFormat 类的对象 sp1 来接收和转换源格式字符串“2005-8-11 18:30:38”，随后又定义了该类的另一个对象 sp2 来接收 sdf1 转换成的 Date 类的对象，并按 sp2 所定义的格式转换成字符串。

在这个过程中，已经实现了利用 SimpleDateFormat 类来把一个字符串转换成 Date 类对象及把一个 Date 对象按用户指定的格式输出的两个功能。

13.6 Math 与 Random 类

▶ 本节视频教学录像：20 分钟

Math 类包含了所有用于几何和三角的浮点运算方法，这些方法都是静态的，每个方法的使用都非常简单，读者一看 JDK 文档就能明白。

Random 类是一个随机数产生器，随机数是按照某种算法产生的，一旦用一个初值创建 Random 对象，就可以得到一系列的随机数。但如果用相同的初值创建 Random 对象，得到的随机数序列是相同的，也就是说，在程序中看到的“随机数”是固定的那些数，起不到“随机”的作用。针对这个问题，Java 设计者在 Random 类的 Random()构造方法中使用当前的时间来初始化 Random 对象，因为没有任何时刻的时间是相同的，所以就可以减少随机数序列相同的可能性。

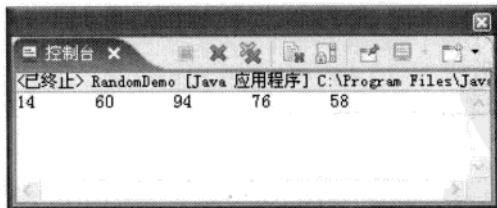
下面的程序就是利用 Random 类来产生 5 个 0~100 之间的随机整数。

【范例 13-6】 利用 Random 类来产生 5 个 0~100 之间的随机整数（代码 13-6. java）。

```
01 import java.util.Random;
02 public class RandomDemo
03 {
04     public static void main(String[] args)
05     {
06         Random r = new Random();
07         for(int i=0;i<5;i++)
08         {
09             System.out.print(r.nextInt(100)+"\t");
10         }
11     }
12 }
```

【运行结果】

保存并运行程序，结果如图所示。



13.7 hashCode()方法

在前面已经学习过 Object 类中的两个方法：equals()、toString()方法。实际上在改写 equals()方法时，也应该去考虑改写 Object 类中的 hashCode()方法。下面先来看一下不改写 hashCode()

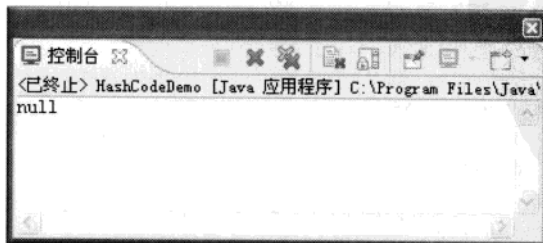
方法时的情况。

【范例 13-7】 hashCode() 方法的使用 (代码 13-7. java)。

```
01 import java.util.* ;
02 class Person
03 {
04     private String name ;
05     private int age ;
06     Person(String name,int age)
07     {
08         this.name = name ;
09         this.age = age ;
10     }
11     public String toString()
12     {
13         return "姓名: "+this.name+" , 年龄: "+this.age ;
14     }
15 }
16 public class HashCodeDemo
17 {
18     public static void main(String args[])
19     {
20         HashMap hm = new HashMap() ;
21         hm.put(new Person("张三",20),"张三") ;
22         System.out.println(hm.get(new Person("张三",20))) ;
23     }
24 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

运行这个程序，读者应该觉得会输出“张三”，但它实际上返回了一个 null 值，其原因就是因为没有改写一个 hashCode() 方法。读者可以将下面的代码加入到程序之中。

```
//注意：这里为了说明问题，只是返回了一个 true
public boolean equals(Object obj)
{
    return true ;
}
public int hashCode()
{
    return 20 ;
}
```

将上面的代码加入到 `HashCodeDemo.java` 程序之中，运行之后可以看到，程序得到了预期的结果：“张三”。到这里，读者应该已经清楚了 `hashCode()` 的含义，在用于存取散列表的时候使用。但是上面修改后的程序中也有一个问题，就是所有的 `Person` 类的对象都拥有同一个散列码，这样在实际中是不可取的。而散列码的取得，也是根据实际的情况而计算出来的，换句话说，只要保证不同的对象有不同的散列码即可。

13.8 对象克隆

本节视频教学录像：51 分钟

“对象克隆 (clone)” 实际上是指将对象重新复制一份。在 Java 里提到 clone 技术，就不能不提 `java.lang.Cloneable` 接口和含有 clone 方法的 `Object` 类。所有具有 clone 功能的类都有一个特性，那就是它直接或间接地实现了 `Cloneable` 接口。否则，在尝试调用 `clone()` 方法时，将会触发 `CloneNotSupportedException` 异常。

那么该如何实现对象的克隆呢？

1. 实现 Cloneable 接口

通过之前的介绍，读者应该知道一个类若要具备 clone 功能，就必须实现 `Cloneable` 接口。做到这一步，clone 功能已经基本实现了。clone 功能对开发者来说，最主要的还是要能够使用它。那么如何才能使用 clone 功能呢？答案是改写 `Object` 类中的 `clone()` 方法。

2. 改写 Object 类中的 clone() 方法

为什么需要改写 `Object` 类中的 `clone()` 方法？这里得再次从 JDK 源码说起。JDK 中 `Object` 类中 `clone()` 方法的声明是：

```
protected native Object clone() throws CloneNotSupportedException;
```

是否注意到，在这里 `clone()` 方法修饰符是 `protected`，而不是 `public`。这种访问的不可见性使得用户对 `clone()` 方法不可见。相信读者已明白为什么要改写 `clone()` 方法了。而且，改写的方法的修饰符必须是 `public`，如果还保留为 `protected`，改写将变得没有实际意义。

【范例 13-8】 对象克隆（代码 13-8.java）。

```
01 class Employee implements Cloneable
```

```

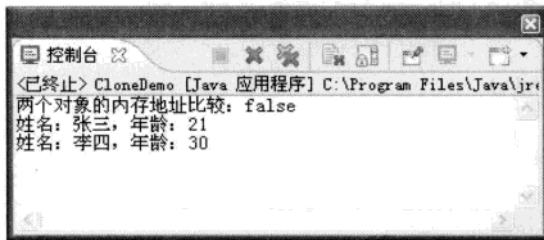
02  {
03      private String name;
04      private int age;
05      public Employee(String name, int age)
06      {
07          this.name = name;
08          this.age = age;
09      }
10      public Object clone() throws CloneNotSupportedException
11      {
12          return super.clone();
13      }
14      public String toString()
15      {
16          return "姓名: " + this.name + ", 年龄: " + this.age;
17      }
18      public int getAge()
19      {
20          return age;
21      }
22      public void setAge(int age)
23      {
24          this.age = age;
25      }
26      public String getName()
27      {
28          return name;
29      }
30      public void setName(String name)
31      {
32          this.name = name;
33      }
34  }
35
36  public class CloneDemo
37  {
38      public static void main(String args[])
39      {
40          Employee e1 = new Employee("张三", 21);
41          Employee e2 = null;
42          try {

```

```
43         e2 = (Employee) e1.clone();
44     } catch (CloneNotSupportedException e) {
45         e.printStackTrace();
46     }
47     e2.setName("李四");
48     e2.setAge(30);
49     System.out.println("两个对象的内存地址比较: " + (e1 == e2));
50     System.out.println(e1);
51     System.out.println(e2);
52 }
53 }
```

【运行结果】

保存并运行程序，结果如图所示。



13.9 练一练

一、填空题

1. 在 Java 中，用_____类来包装整数。
2. System 类中的所有成员都是_____的，引用这些变量和方法时，直接使用_____类名作为前缀。
3. 在 Java 中启动一个新的进程，即执行其他程序时使用_____方法。

二、简答题

简述如何实现对象的克隆。

13.10 跟我上机

编写一段程序，使程序运行后能自动打开计算器。

第 14 章

包及访问权限



本章视频教学录像：43 分钟

包是类的一种特殊的性质，包的作用在管理大型的项目时会变得更加明显，使用包能更合理地管理大量的类文件，可以设置他人对类成员的访问权等。本章介绍包的基本概念、import语句的使用、类成员访问权限的控制，以及Jar命令的使用等。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握包的基本概念
- ☐ 掌握 import 语句的使用
- ☐ 了解 JDK 中常见的包
- ☐ 熟悉类成员的访问控制权限
- ☐ 了解 Java 的命名习惯
- ☐ 了解 Jar 命令的使用



在用 Java 开发大型项目时，通常要把类分门别类地存到文件里，再将这些文件一起编译执行，这样的程序代码将更易于维护。同时在将类分割开之后，对于类的使用也就有了相应的访问权限。

14.1 包的概念及使用

本节视频教学录像：22 分钟

本节介绍包的基本概念、import 语句的使用及 JDK 中常见的包。

14.1.1 包（package）的基本概念

当一个大型程序由数个不同的组别或人员共同开发时，用到相同的类名称是很有可能的事。如果这种情况发生，还要确保程序可以正确运行，就必须通过 package 关键字来帮忙了。

package 是在使用多个类或接口时，为了避免名称重复而采用的一种措施。那么具体应该如何使用呢？在类或接口的最上面一行加上 package 的声明就可以了。package 的声明如下。

```
package package 名称；
```

经过 package 的声明之后，同一文件内的接口或类就都会被纳入相同的 package 中。

【范例 14-1】 package 的使用（代码 14-1. java）。

```
01 package demo.java ;
02 class Person
03 {
04     public String talk()
05     {
06         return "Person —— >> talk()";
07     }
08 }
09
10 class TestPackage1
11 {
12     public static void main(String[] args)
13     {
14         System.out.println(new Person().talk());
15     }
16 }
```

【代码详解】

除了第 1 行加的 package demo.java 声明语句之外，其余的程序都是读者见过的。由于在第 1 行声明了一个 demo.java 的包，所以就相当于将 Person 类、TestPackage1 类放入了 demo.java

文件夹之下。

现在来看一下上面的程序是如何编译的。

```
javac -d . TestPackage1.java
```

“-d”：表示生成目录。

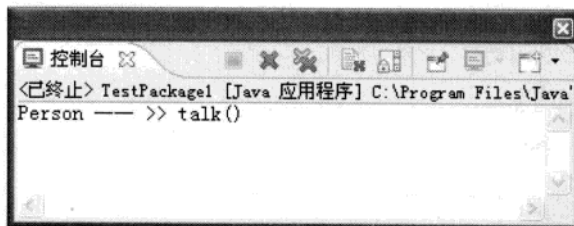
“.”：表示在当前目录下生成。

这样就会在当前目录下生成一个 demo 文件夹,在 demo 文件夹下又会生成一个 java 文件夹,在此文件夹下会有编译好的 Person.class 和 TestPackage1.class,编译好之后用下面的语法来执行它。

```
java demo.java.TestPackage1
```

【运行结果】

保存并运行程序,结果如图所示。



14.1.2 import 语句的使用

到目前为止,所介绍的类都是属于同一个 package 的,因此在程序代码的编写上并不需要做修改。但如果几个类分别属于不同的 package,在某个类要访问到其他类的成员时,则必须做下列的修改。

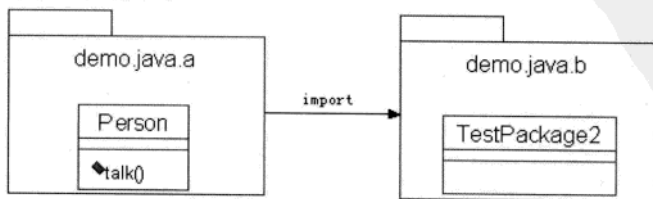
若某个类需要被访问时,则必须把这个类公开出来,也就是说,此类必须声明成 public。

若要访问不同 package 内某个 public 类的成员时,在程序代码内必须明确地指明“被访问 package 的名称.类名称”。package 的导入如下。

```
import package 名称.类名称 ;
```

通过 import 命令,可将某个 package 内的整个类导入,因此后续的程序代码便不用再写上被访问 package 的名称了。

下面用一个范例来说明 import 命令的用法。此范例与 TestPackage1 类似,只是将两个类分别放在了不同的包中,如图所示。



【范例 14-2】 Person 类的声明（代码 14-2. java）。

```
01 package demo.java.a ;
02
03 public class Person
04 {
05     public String talk()
06     {
07         return "Person —— >> talk()";
08     }
09 }
```

【代码详解】

第 1 行声明了一个 demo.java.a 的包，将 Person 类放入此包之中。

【范例 14-3】 包的导入使用范例 1（代码 14-3. java）。

```
01 package demo.java.b ;
02 import demo.java.a.Person ;
03
04 class TestPackage2
05 {
06     public static void main(String[] args)
07     {
08         System.out.println(new Person().talk());
09     }
10 }
```

【代码详解】

第 1 行声明了一个 demo.java.b 包，将 TestPackage2 类放入此包之中。

第 2 行使用 import 语句，将 demo.java.a 包中的 Person 类导入到此包之中。



提示：可以将第 2 行的 import demo.java.a.Person 改成 import demo.java.a.*，表示导入包中的所有类。在 Java 中有这样的规定：导入全部类或是导入指定的类，对于程序的性能是没有影响的，所以在开发中直接导入全部类会比较方便。

另外，上面的程序也可以写成如下的形式。

【范例 14-4】 包的导入使用范例 2（代码 14-4. java）。

```
01 package demo.java.b ;
```

```

02
03  class TestPackage3
04  {
05      public static void main(String[] args)
06      {
07          System.out.println(new demo.java.a.Person().talk());
08      }
09  }

```

【范例分析】

可以看到，在程序中并没有使用 import 语句，但是在第 7 行使用 Person 类的时候使用了“包名.类名”的方式，所以在程序中也可以用此方法来使用非本类所在的包中的类。

14.1.3 JDK 中常见的包

SUN 公司在 JDK 中为程序开发者提供了各种实用类，这些类按功能不同分别被放入了不同的包中，供开发者使用。下面简要介绍其中最常用的几个包。

(1) java.lang —— 包含一些 Java 语言的核心类，如 String、Math、Integer、System 和 Thread，提供常用功能。在 java.lang 包中还有一个子包：java.lang.reflect，用于实现 java 类的反射机制。

(2) java.awt —— 包含构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。

(3) javax.swing —— 此包用于建立图形用户界面，包中的组件相对于 java.awt 包而言是轻量级组件。

(4) java.applet —— 包含 applet 运行所需的一些类。

(5) java.net —— 包含执行与网络相关的操作的类。

(6) java.io —— 包含能提供多种输入/输出功能的类。

(7) java.util —— 包含一些实用工具类，如定义系统特性、与日期日历相关的方法。



提示：在 Java1.2 以后的版本中，java.lang 这个包会自动被导入，对于其中的类，不再需要使用 import 语句来做导入，如前面经常使用的 System 类。

14.2 类成员的访问控制权限

▶ 本节视频教学录像：9 分钟

在 JAVA 中有 4 种访问控制权限，分别为 private、default、protected、public。

1. private 访问控制符

在前面已经介绍了 private 访问控制符的作用，如果一个成员方法或成员变量名的前面使用了 private 访问控制符，那么这个成员只能在这个类的内部使用。



提示：不能在方法体内声明的变量前面加 `private` 修饰符。

2. default 默认访问控制符

如果一个成员方法或成员变量名前面没有使用任何访问控制符,就称这个成员所拥有的是默认的 (default) 访问控制符。默认的访问控制成员可以被这个包中的其他类访问。如果一个子类与其父类位于不同的包中,子类也不能访问父类中的默认访问控制成员。

3. protected 访问控制符

如果一个成员方法或成员变量名前面使用了 `protected` 访问控制符,那么这个成员既可以被同一个包中的其他类访问,也可以被不同包中的子类访问。

4. public 访问控制符

如果一个成员方法或成员变量名前面使用了 `public` 访问控制符,那么这个成员可以被所有的类访问,不管访问类与被访问类是否在同一个包中。

最后,用下表来总结上述访问控制符的权限。

	private	default	protected	public
同一类	✓	✓	✓	✓
同一包中的类		✓	✓	✓
不同包的子类			✓	✓
其他包中的类				✓

下面的范例介绍了 `protected` 关键字的使用方法。

【范例 14-5】 `protected` 关键字的使用 (代码 14-5. java)。

```

01 package demo.java.a ;
02
03 public class Person
04 {
05     protected String name ;
06     public String talk()
07     {
08         return "Person—— >> talk()" ;
09     }
10 }
```

【代码详解】

第 5 行声明一个 `String` 类型的属性 `name`,该变量用 `protected` 关键字声明,所以此属性只能在本类及其子类中使用。

【范例 14-6】 类成员的访问控制权限使用范例 1（代码 14-6. java）。

```
01 package demo.java.b ;
02 import demo.java.a.* ;
03
04 public class Student extends Person
05 {
06     public Student(String name)
07     {
08         this.name = name ;
09     }
10     public String talk()
11     {
12         return "Person —— >> talk() , "+this.name ;
13     }
14 }
```

【代码详解】

第 2 行导入 Person 类。

第 4 行 Student 类继承自 Person 类。

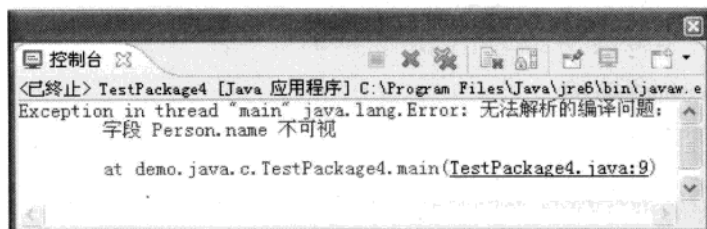
第 8 行 Student 类访问 Person 类中的 name 属性。

【范例 14-7】 类成员的访问控制权限使用范例 2（代码 14-7. java）。

```
01 package demo.java.c ;
02 import demo.java.b.* ;
03
04 class TestPackage4
05 {
06     public static void main(String[] args)
07     {
08         Student s = new Student("javafans") ;
09         s.name = "javafans" ;
10         System.out.println(s.talk()) ;
11     }
12 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

可以看到，在第 9 行通过对象调用受保护的属性，所以程序在编译时 JDK 会报错。

14.3 Java 的命名习惯

本节视频教学录像：2 分钟

读者通过查看 JDK 文档，可以看到，JDK 中类的声明、方法的声明、常量的声明等都是有一定规律的。

- (1) 包名中的字母一律小写，如 demo.java。
- (2) 类名、接口名应当使用名词，每个单词的首字母大写，如 TestPerson。
- (3) 变量名（属性名）第 1 个单词小写，后面的每个单词首字母大写，如 newLxh。
- (4) 方法名的第 1 个单词小写，后面每个单词的首字母大写，如 talkMySelf()。
- (5) 常量名中的每个字母一律大写，如 COUNTRY。

14.4 打包工具——Jar 命令的使用

本节视频教学录像：10 分钟

开发者用的 JDK 中的包和类，主要位于 JDK 的安装目录下的 jre\lib\rt.jar 文件中。由于 Java 虚拟机会自动找到这个 jar 包，所以在开发中使用这个 jar 包的类时，无需再用 classpath 指向它们的位置。

jar 文件就是 Java Archive File，而它的应用是与 Java 息息相关的。jar 文件就是一种压缩文件，与常见的 ZIP 压缩文件格式兼容，习惯上称之为 jar 包。如果开发者开发了许多类，当需要把这些类提供给用户使用，通常都会将这些类压缩到一个 jar 文件中，以 jar 包的方式提供给用户使用。只要别人的 classpath 环境变量的设置中包含这个 jar 文件，Java 虚拟机就能自动在内存中解压这个 jar 文件，把这个 jar 文件当做一个目录，在这个 jar 文件中去寻找所需要的类名及包名所对应的目录结构。

jar 命令是随 JDK 一起安装的，存放在 JDK 安装目录下的 bin 目录中，Windows 下的文件名为 jar.exe，Linux 下的文件名为 jar。jar 命令是 Java 中提供的一个非常有用的命令，可以用来对大量的类（.class 文件）进行压缩，然后存为 jar 文件。那么通过 jar 命令所生成的 jar 压缩文件又有什么优点呢？一方面，可以方便管理大量的类文件；另一方面，进行了压缩也减少了文件所占的空间。对于用户来说，除了安装 JDK 之外什么也不需要，因为 SUN 公司已经帮开发者做好了其他的一切工作。

在命令行窗口下运行 jar.exe 程序，就可以看到下图所示的界面。



从 JDK 的提示中可以看到,只要在命令行中使用以下命令就可以将一个包打成一个 jar 文件。

```
jar -cvf create.jar demo
```

-c: 创建新的存档。

-v: 生成详细输出到标准输出上。

-f: 指定存档文件名。

create.jar: 是生成 jar 文件的名称。

demo: 要打成 jar 文件的包。

14.5 练一练

一、填空题

1. 声明一个名称为“baohan”的包的语句为_____。
2. 在“javac -d .TestPackage1.java”中,参数“-d”的意义为_____。
3. 在声明一个类时,如果需要它被别的类访问,则需要将其声明为_____类型。
4. 在 Java 中,默认的访问控制符为_____。

二、简答题

简述 Java 中的命名习惯。

14.6 跟我上机

编写一段程序,声明一个包,在另一个包中使用 import 语句访问使用。

第 15 章

异常处理



本章视频教学录像：**43 分钟**

不管使用的是哪种语言进行程序设计，都会产生各种各样的错误。Java提供有强大的异常处理机制。在Java中，所有的异常被封装到一个类中，程序出错时会将异常抛出。本章讲解Java中异常的基本概念、对异常的处理、异常的抛出，以及怎样编写自己的异常类。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握异常的基本概念
- ☐ 掌握对异常的处理机制
- ☐ 熟悉如何在程序和方法中抛出异常
- ☐ 了解如何编写自己的异常类



即使在编译时没有错误信息产生,但在程序运行时,经常会出现一些运行时的错误,这种错误对 Java 而言是一种异常。有了异常就要有相应的处理方式。

15.1 异常的基本概念

▶ 本节视频教学录像: 3 分钟

异常也称为例外,是在程序运行过程中发生的、会打断程序正常执行的事件。下面是几种常见的异常。

- (1) 算术异常 (ArithmeticException)。
- (2) 没有给对象开辟内存空间时会出现空指针异常 (NullPointerException)。
- (3) 找不到文件异常 (FileNotFoundException)。

所以在程序设计时,必须考虑到可能发生的异常事件,并做出相应的处理,这样才能保证程序可以正常运行。

Java 的异常处理机制也秉承着面向对象的基本思想。在 Java 中,所有的异常都是以类的类型存在。除了内置的异常类之外,Java 也可以自定义异常类。此外,Java 的异常处理机制也允许自定义抛出异常。

15.1.1 为何需要异常处理

异常是在程序运行过程中发生的事件,比如除 0 溢出、数组越界、文件找不到等,这些事件的发生将阻止程序的正常运行。为了加强程序的健壮性,在程序设计时,必须考虑到可能发生的异常事件,并做出相应的处理。在 C 语言中,可通过使用 if 语句来判断是否出现了异常,同时,可通过被调用函数的返回值感知在其中产生的异常事件,并进行处理。全程变量 ErrNo 常常用来反映一个异常事件的类型。但是,这种错误处理机制会导致不少问题。

Java 通过面向对象的方法来处理异常。在一个方法的运行过程中,如果发生了异常,则这个方法生成代表该异常的一个对象,并把它交给运行时系统,运行时系统寻找相应的代码来处理这一异常。我们把生成异常对象并把它提交给运行时系统的过程称为抛弃(throw)一个异常。运行时系统在方法的调用栈中查找,从生成异常的方法开始进行回溯,直到找到包含相应异常处理的方法为止,这一个过程称为捕获(catch)一个异常。

15.1.2 简单的异常范例

Java 本身已有相当好的机制来处理异常的发生。下面先来看看 Java 是如何处理异常的。TestException_1 是一个错误的程序,它在访问数组时,下标值已超过了数组下标所容许的最大值,因此会有异常发生。

【范例 15-1】 简单的异常范例(代码 15-1. java)。

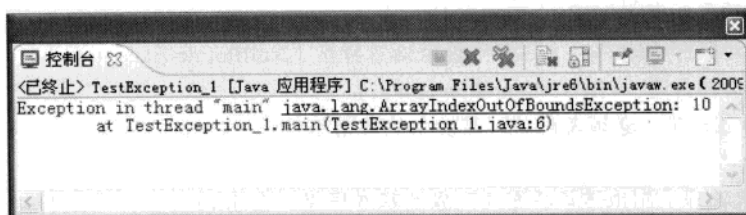
```
01 public class TestException_1
02 {
03     public static void main(String args[])
```



```

04    {
05        int arr[]=new int[5];           // 容许 5 个元素
06        arr[10]=7;                     // 下标值超出所容许的范围
07        System.out.println("end of main() method !!");
08    }
09 }
    
```

在编译的时候程序不会发生任何错误，但是在执行到第 6 行时，就会产生如图所示的错误信息。



错误的原因在于数组的下标值超出了最大允许的范围。Java 发现这个错误之后，便由系统抛出“ArrayIndexOutOfBoundsException”这个种类的异常，用来表示错误的原因，并停止运行程序。如果没有编写相应的处理异常的代码，Java 的默认异常处理机制会先抛出异常，然后停止运行程序。

15.1.3 异常的处理

TestException_1 的异常发生后，Java 便把这个异常抛了出来，可是抛出来之后没有程序代码去捕捉它，所以程序到第 6 行便结束，因此根本不会执行到第 7 行。如果加上捕捉异常的代码，则可针对不同的异常做妥善的处理，这种处理的方式称为异常处理。

异常处理是由 try、catch 与 finally 等 3 个关键字所组成的程序块，其语法如下。

```

try{
    要检查的程序语句 ;
    ...
}catch(异常类 对象名称){
    异常发生时的处理语句 ;
}finally{
    一定会运行到的程序代码 ;
}
    
```

语法是依据下列的顺序来处理异常。

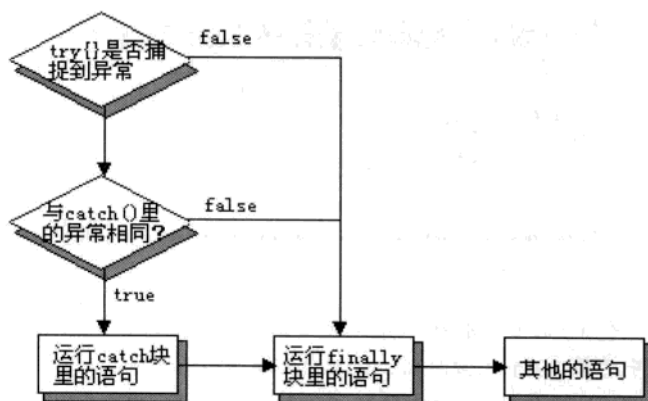
- (1) try 程序块若有异常发生，程序的运行便中断，并抛出“异常类所产生的对象”。
- (2) 抛出的对象如果属于 catch()括号内欲捕获的异常类，catch 则会捕捉此异常，然后进到 catch 的块里继续运行。
- (3) 无论 try 程序块是否捕捉到异常，或者捕捉到的异常是否与 catch()括号里的异常相同，最后一定会运行 finally 块里的程序代码。

finally 的程序代码块运行结束后，程序再回到 try-catch-finally 块之后继续执行。

由上述的过程可知，在异常捕捉的过程中做了两个判断：第 1 个是 try 程序块是否有异常产生，第 2 个是产生的异常是否和 catch() 括号内欲捕捉的异常相同。

值得一提的是，finally 块是可以省略的。如果省略了 finally 块不写，那么在 catch() 块运行结束后，程序将跳到 try-catch 块之后继续执行。

根据这些基本概念与运行的步骤，可以绘制出下图所示的流程。



在上述格式中，“异常类”指的是由程序抛出的对象所属的类，例如 TestException_1 中出现的“ArrayIndexOutOfBoundsException”就是属于异常类的一种。至于有哪些异常类以及它们之间的继承关系，稍后会做进一步的探讨。下面的程序代码加入了 try 与 catch，使得程序本身具有了捕捉异常与处理异常的能力。

【范例 15-2】 异常处理的使用（代码 15-2. java）。

```

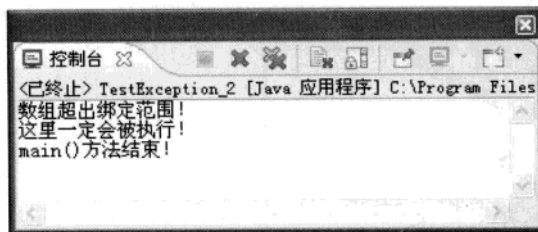
01  public class TestException_2
02  {
03      public static void main(String args[])
04      {
05          try                                // 检查这个程序块的代码
06          {
07              int arr[]=new int[5];
08              arr[10]=7;                     // 在这里会出现异常
09          }
10          catch(ArrayIndexOutOfBoundsException e)
11          {
12              System.out.println("数组超出绑定范围！");
13          }
14          finally                            // 这个块的程序代码一定会执行
15          {
16              System.out.println("这里一定会被执行！");
17          }

```

```
18         System.out.println("main()方法结束!");
19     }
20 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 7 行声明了一个 arr 的整型数组，并开辟了 5 个数据空间。

第 8 行为数组中的第 10 个元素赋值，此时已经超出了该数组本身的范围，所以会出现异常。发生异常之后，程序语句转到 catch 语句中去处理，程序通过 finally 代码块统一结束。

【范例分析】

程序的第 5~9 行的 try 块是用来检查是否会有异常发生的。若有异常发生，且抛出的异常是属于 `ArrayIndexOutOfBoundsException` 类，则会运行第 10~13 行的代码块。因为第 8 行所抛出的异常正是 `ArrayIndexOutOfBoundsException` 类，因此第 12 行会输出“数组超出绑定范围!”字符串。由本例可看出，通过异常的机制，即使程序运行时发生问题，只要能捕捉到异常，程序便能顺利地运行到最后，而且还能适时地加入对错误信息的提示。

程序 `TestException_2` 里的第 10 行，如果程序捕捉到了异常，则在 catch 括号内的异常类 `ArrayIndexOutOfBoundsException` 之后生成一个对象 `e`，利用此对象可以得到异常的相关信息。下例说明了类对象 `e` 的应用。

【范例 15-3】 类对象 e 的使用（代码 15-3. java）。

```
01 public class TestException_3
02 {
03     public static void main(String args[])
04     {
05         try
06         {
07             int arr[]=new int[5];
08             arr[10]=7;
09         }
10         catch(ArrayIndexOutOfBoundsException e){
11             System.out.println("数组超出绑定范围!");
```

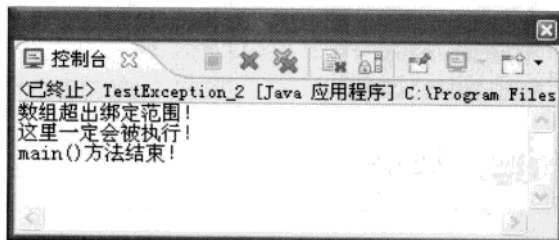
```

12         System.out.println("异常: "+e);           // 显示异常对象 e 的内容
13     }
14     System.out.println("main()方法结束! ");
15 }
16 }

```

【运行结果】

保存并运行程序，结果如图所示。

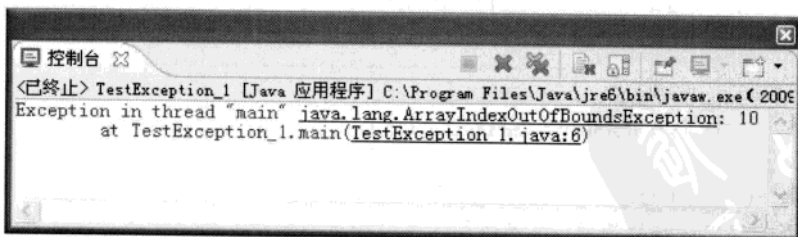


【范例分析】

范例 TestException_3 省略了 finally 块，但程序依然可以运行。在第 10 行中，把 catch() 括号内的内容想象成是方法的参数，而 e 就是 ArrayIndexOutOfBoundsException 类的对象。对象 e 接收到由异常类所产生的对象之后，就进到第 11 行，输出“数组超出绑定范围！”这一字符串，第 12 行则是输出异常所属的种类，也就是 java.lang. ArrayIndexOutOfBoundsException，而 java.lang 正是 ArrayIndexOutOfBoundsException 类所属的包。

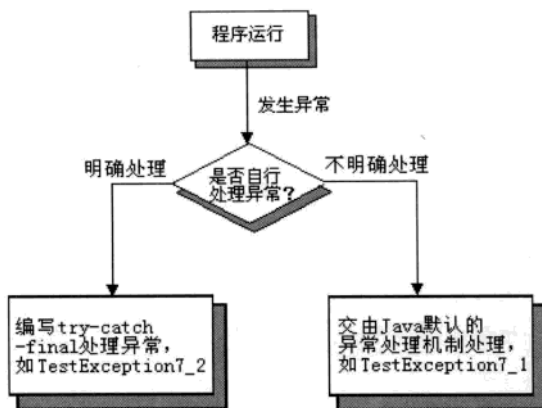
15.1.4 异常处理机制的回顾

当异常发生时，通常可以用两种方法来处理，一种是交由 Java 默认的异常处理机制做处理。但这种处理方式，Java 通常只能输出异常信息，接着便终止程序的运行。如 TestException_1 的异常发生后，Java 默认的异常处理机制的显示如图所示。



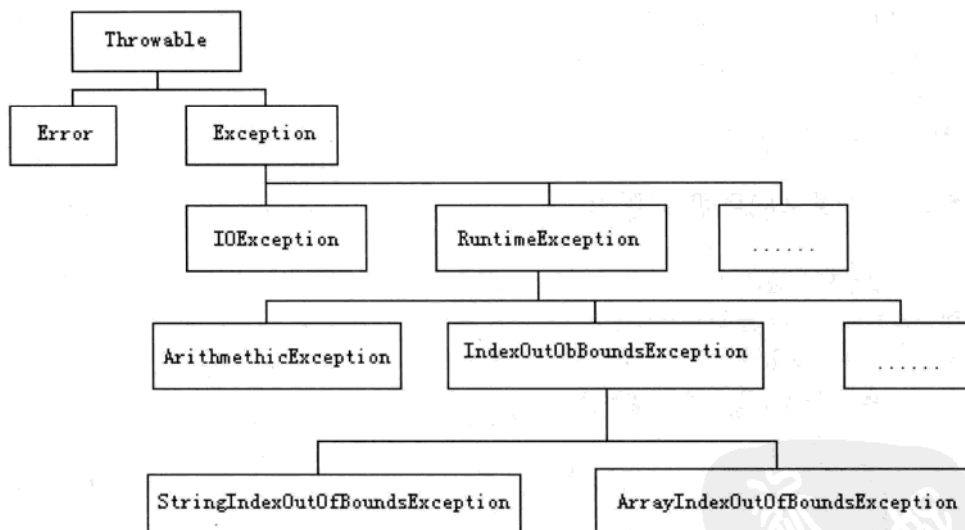
接着结束 TestException_1 的运行。

另一种处理方式是用自行编写的 try-catch-finally 块来捕捉异常，如 TestException_2 与 TestException_3。自行编写程序代码来捕捉异常最大的好处是：可以灵活操控程序的流程，且可做出最适当的处理。下图绘出了异常处理机制的选择流程。



15.2 异常类的继承架构

异常可分为两大类：`java.lang.Exception` 类与 `java.lang.Error` 类。这两个类均继承自 `java.lang.Throwable` 类。下图为 `Throwable` 类的继承关系图。



习惯上将 `Error` 与 `Exception` 类统称为异常类，但这两者本质上还是不同的。`Error` 类专门用来处理严重影响程序运行的错误，可是通常程序设计者不会设计程序代码去捕捉这种错误，其原因在于即使捕捉到它，也无法给予适当的处理，如 `JAVA` 虚拟机出错就属于一种 `Error`。

不同于 `Error` 类，`Exception` 类包含了一般性的异常，这些异常通常在捕捉到之后便可做妥善的处理，以确保程序继续运行，如 `TestException_2` 里所捕捉到的 `ArrayIndexOutOfBoundsException` 就是属于这种异常。

从异常类的继承架构图中可以看出，`Exception` 类扩展出了数个子类，其中 `IOException` 和 `RuntimeException` 是较常用的两种。`RuntimeException` 即使不编写异常处理的程序代码，依然可以编译成功，而这种异常必须是在程序运行时才有可能发生，例如数组的索引值超出了范围。

与 `RuntimeException` 不同的是, `IOException` 一定要编写异常处理的程序代码才行, 它通常用来处理与输入/输出相关的操作, 如文件的访问、网络的连接等。

当异常发生时, 发生异常的语句代码会抛出一个异常类的实例化对象, 之后此对象与 `catch` 语句中的类的类型进行匹配, 然后在相应的 `catch` 中进行处理。

15.3 抛出异常

▶ 本节视频教学录像: 37 分钟

前两节介绍了 `try-catch-finally` 程序块的编写方法, 本节介绍如何抛出 (`throw`) 异常, 以及如何由 `try-catch` 来接收所抛出的异常。抛出异常的方式有以下两种。

- (1) 程序中抛出异常。
- (2) 指定方法抛出异常。

15.3.1 在程序中抛出异常

在程序中抛出异常时, 一定要用到 `throw` 这个关键字, 其语法如下。

```
throw 异常类实例对象 ;
```

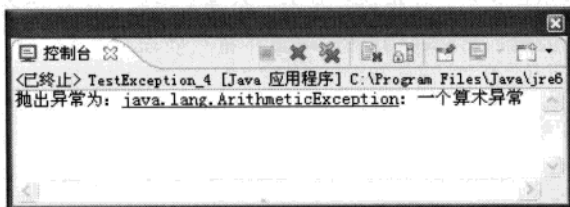
从上述格式中可以看到, 在 `throw` 后面抛出的是一个异常类的实例对象。下面来看一个实例。

【范例 15-4】 在程序中抛出异常 (代码 15-4. java)。

```
01 public class TestException_4
02 {
03     public static void main(String args[])
04     {
05         int a=4,b=0;
06         try
07         {
08             if(b==0)
09                 throw new ArithmeticException("一个算术异常"); // 抛出异常
10             else
11                 System.out.println(a+"/"+b+"="+a/b); // 若抛出异常, 则执行此行
12         }
13         catch(ArithmeticException e)
14         {
15             System.out.println("抛出异常为: "+e);
16         }
17     }
18 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

程序 TestException_4 是要计算 a/b 的值。因 b 是除数，因此不能为 0。若 b 为 0，系统则会抛出 ArithmeticException 异常，代表除到 0 这个数。

在 try 块里，利用第 8 行来判断除数 b 是否为 0。如果 $b=0$ ，则运行第 9 行的 throw 语句，抛出 ArithmeticException 异常。如果 b 不为 0，则输出 a/b 的值。在此例中强制把 b 设为 0，因此 try 块的第 9 行会抛出异常，并由第 13 行的 catch() 捕捉到异常。

抛出异常时，throw 关键字所抛出的是异常类的实例对象，因此第 9 行的 throw 语句必须使用 new 关键字来产生对象。

15.3.2 指定方法抛出异常

如果方法内的程序代码可能会发生异常，且方法内又没有使用任何的代码块来捕捉这些异常，则必须在声明方法时一并指明所有可能发生的异常，以便让调用此方法的程序得以做好准备来捕捉异常。也就是说，如果方法会抛出异常，则可将处理此异常的 try-catch-finally 块写在调用此方法的程序代码内。

如果要由方法抛出异常，则方法必须用下面的语法来声明。

方法名称 (参数...) throws 异常类 1, 异常类 2, ...

范例 TestException_5 是指定由方法来抛出异常的，注意此处把 main() 方法与 add() 方法编写在了同一个类内。

【范例 15-5】 使用指定方法抛出异常 (代码 15-5. java)。

```
01  class Test
02  {
03      // throws 在指定方法中不处理异常，在调用此方法的地方处理
04      void add(int a,int b) throws Exception
05      {
06          int c;
07          c=a/b;
08          System.out.println(a+"/"+b+"="+c);
09      }
```

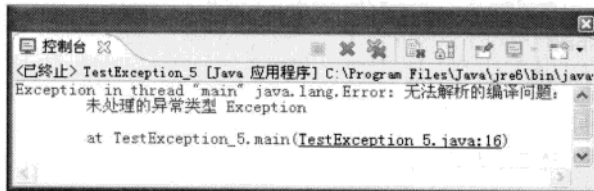
```

10 }
11 public class TestException_5
12 {
13     public static void main(String args[])
14     {
15         Test t = new Test();
16         t.add(4,0);
17     }
18 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 1~10 行声明了一个 Test 类, 此类中有一个 add(int a,int b)方法, 但在此方法后面用 throws 关键字抛出了一个 Exception 异常。

第 15 行实例化了一个 Test 对象 t, 在第 16 行调用 Test 类中的 add()方法。

【范例分析】

从编译结果中可以看到, 如果在类的方法中用 throws 抛出一个异常, 那么在调用它的地方就必须明确地用 try-catch 来捕捉。



提示: 在 TestException_5 程序之中, 如果在 main() 方法的后面再用 throws Exception 声明的话, 那么程序也是依然可以编译通过的。也就是说在调用 throws 抛出异常的方法时, 可以将此异常在方法中再向上传递, 而 main() 方法是整个程序的起点, 所以在 main() 方法处如果再用 throws 抛出异常, 则此异常就将交由 JVM 进行处理。

15.4 编写自己的异常类

▶ 本节视频教学录像: 3 分钟

为了处理各种异常, Java 可通过继承的方式编写自己的异常类。因为所有可处理的异常类均继承自 Exception 类, 所以自定义异常类也必须继承这个类。自己编写异常类的语法如下。

```

class 异常名称 extends Exception
{

```

```
.....  
}
```

读者可以在自定义异常类里编写方法来处理相关的事件,甚至不编写任何语句也可以正常地工作,这是因为父类 `Exception` 已提供相当丰富的方法,通过继承,子类均可使用它们。

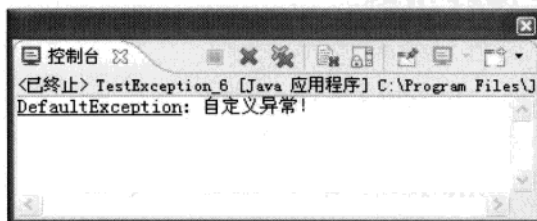
下面用一个范例来说明如何定义自己的异常类,以及如何使用它们。

【范例 15-6】 定义自己的异常类 (代码 15-6. java)。

```
01 class DefaultException extends Exception  
02 {  
03     public DefaultException(String msg)  
04     {  
05         // 调用 Exception 类的构造方法,存入异常信息  
06         super(msg);  
07     }  
08 }  
09 public class TestException_6  
10 {  
11     public static void main(String[] args)  
12     {  
13         try  
14         {  
15             // 在这里用 throw 直接抛出一个 DefaultException 类的实例对象  
16             throw new DefaultException("自定义异常!");  
17         }  
18         catch(Exception e)  
19         {  
20             System.out.println(e);  
21         }  
22     }  
23 }
```

【运行结果】

保存并运行程序,结果如图所示。



【代码详解】

第 1~8 行声明了一个 DefaultException 类，此类继承自 Exception 类，所以此类为自定义异常类。

第 6 行调用 super 关键字，调用父类（Exception）的有一个参数的构造方法，传入的为异常信息。

Exception 构造方法如下。

```
public Exception(String message)
```

第 16 行用 throw 抛出一个 DefaultException 异常类的实例化对象。

【范例分析】

在 JDK 中提供的大量 API 方法之中含有大量的异常类，但这些类在实际开发中往往并不能完全满足设计者对程序异常处理的需要，在这个时候就需要用户自己去定义所需的异常类，用一个类清楚地写出所需要处理的异常。

15.5 练一练

一、填空题

1. 在 Java 中，所有的异常都是以_____存在。
2. 异常处理是指_____。
3. 在程序中抛出异常，一定会使用到的关键字为_____。

二、简答题

1. 在 Java 中，通常用哪两种方法来处理异常？
2. 简述抛出异常的两种方式。

15.6 跟我上机

编写一个对 0 进行除操作的程序，使程序抛出此异常，并输出“被除数为 0，程序出错!”。

第 16 章

Java类集框架



本章视频教学录像：1 小时 49 分钟

Java的类集框架可以使程序处理对象的方法标准化，类集接口是构造类集框架的基础，使用迭代方法访问类集可以使对类集的操作更高效。本章讲解类集接口的有关概念，介绍List接口、集合接口和SortedSet接口的使用，并介绍ArrayList类、HashSet类和TreeSet类的相关知识。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握类集接口的概念
- ☐ 掌握 List 接口、集合接口和 SortedSet 接口
- ☐ 熟悉 Collection 接口中的 ArrayList 类、HashSet 类和 TreeSet 类的方法
- ☐ 了解通过迭代方法访问类集

Java 的类集 (Collection) 框架可以使程序处理对象组的方法标准化。在 Java 2 出现之前, Java 提供一些专门的类, 如 Dictionary、Vector、Stack 和 Properties 等去存储和操作对象组。尽管这些类非常有用, 但它们却缺少一个集中、统一的主题。因此使用 Vector 的方法就会与使用 Properties 的方法不同。以前的专门的方法也没有被设计成易于扩展和能适应新的环境的形式。而类集解决了这些 (以及其他的一些) 问题。类集框架被设计成拥有以下几个特性: 第一, 这种框架是高性能的。对基本类集 (动态数组、链接表、树和散列表等) 的实现是高效率的, 一般很少需要人工去对这些“数据引擎”编写代码。第二, 框架必须允许不同类型的类集以相同的方式和高度互操作方式工作。第三, 类集必须是容易扩展和修改的。为了实现这一目标, 类集框架被设计成包含了一组标准接口。为这些接口提供了几个标准的实现工具 (例如 LinkedList、HashSet 和 TreeSet 等), 通常就是这样使用的。如果读者愿意的话, 也可以实现自己的类集。

为了方便起见, 可创建用于各种特殊目的的实现工具, 一部分工具可以使自己的类集实现更加容易。最后, 增加了允许将标准数组融合到类集框架中的机制。

算法 (Algorithms) 是类集机制的另一个重要部分。算法操作类集, 它在 Collections 类中被定义为静态方法。因此它们可以被所有的类集所利用。每一个类集不必实现它自己的方案, 算法提供了一个处理类集的标准方法。

由类集框架创建的另一项是 Iterator 接口。一个迭代程序 (iterator) 提供了一个多用途的、标准化的方法, 用于每次访问类集的一个元素, 因此迭代程序提供了一种枚举类集内容 (enumerating the contents of a collection) 的方法。因为每一个类集都实现 Iterator 接口, 所以通过由 Iterator 定义的方法, 任一类集的元素都能被访问到。因此稍作修改, 循环通过集合的程序代码也可以被用来循环通过列表。

除了类集之外, 框架还定义了几个映射接口和类。映射 (Maps) 存储键值到值的对应。尽管映射在对象的正确使用上不是“类集”, 但它们完全用类集集成。在类集框架的语言中, 可以获得映射的类集“视图” (collection-view), 这个“视图”包含了从存储在类集中的映射得到的元素。因此如果选择了一个映射, 就可以将其当做一个类集来处理。

对于由 java.util 定义的原始类, 类集机制被更新以便它们也能够集成到新的系统里。所以理解下面的说法是很重要的: 尽管类集的增加改变了许多原始工具类的结构, 但它却不会导致原始的工具类被抛弃。类集仅仅是提供了处理事情的一个更好的方法。

16.1 类集接口

本节视频教学录像: 35 分钟

类集框架定义了几个接口。本节对每一个接口进行概述。首先讨论类集接口是因为它们决定了 collection 类的基本特性。不同的是, 具体类仅仅是提供了标准接口的不同实现。支持类集的接口总结在下表中。

接口	描述
Collection	能操作对象组, 它位于类集层次结构的顶层
List	扩展 Collection 去处理序列 (对象的列表)
Set	扩展 Collection 去处理集合, 集合必须包含唯一元素
SortedSet	扩展 Set 去处理排序集合

除了类集接口之外, 类集也使用 Comparator、Iterator 和 ListIterator 等接口。关于这些接口

将在本章后面做更深入的讲解。简单地说, Comparator 接口定义了两个对象比较方法, 即 Iterator 和 ListIterator 接口类集中的对象。

为了在它们的使用中提供最大的灵活性, 类集接口允许对一些方法进行选择。可选择的方法使得使用者可以更改类集的内容。支持这些方法的类集被称为可修改的 (modifiable)。不允许修改其内容的类集被称为不可修改的 (unmodifiable)。而所有内置的类集都是可修改的。如果对一个不可修改的类集使用这些方法, 将引发一个 UnsupportedOperationException 异常。

16.1.1 类集接口

Collection 接口是构造类集框架的基础, 它声明所有类集都将拥有的核心方法, 这些方法被总结在下表中。因为所有类集实现 Collection, 所以熟悉它的方法对于清楚地理解框架是必要的。其中有几种方法可能会引发一个 UnsupportedOperationException 异常。正如上面解释的那样, 这些将发生在类集不能被修改的时候。当一个对象与另一个对象不兼容, 例如企图增加一个不兼容的对象到一个类集中时, 将产生一个 ClassCastException 异常。

方法	描述
boolean add(Object obj)	将obj加入到调用类集中。如果obj被加入到类集中了, 则返回true; 如果obj已经是类集中的一个成员或类集不能被复制时, 则返回false
boolean addAll(Collection c)	将c中的所有元素都加入到调用类集中, 如果操作成功(元素被加入了), 则返回true, 否则返回false
void clear()	从调用类集中删除所有元素
boolean contains(Object obj)	如果obj是调用类集的一个元素, 则返回true, 否则返回false
boolean containsAll(Collection c)	如果调用类集包含了c中的所有元素, 则返回true, 否则返回false
boolean equals(Object obj)	如果调用类集与obj相等, 则返回true, 否则返回false
int hashCode()	返回调用类集的散列码
boolean isEmpty()	如果调用类集是空的, 则返回true, 否则返回false
Iterator iterator()	返回调用类集的迭代程序
Boolean remove(Object obj)	从调用类集中删除obj的一个实例。如果这个元素被删除了, 则返回true, 否则返回false
Boolean removeAll(Collection c)	从调用类集中删除c的所有元素。如果类集被改变了(也就是说元素被删除了), 则返回true, 否则返回false
Boolean retainAll(Collection c)	删除调用类集中除了包含在c中的元素之外的全部元素。如果类集被改变了(也就是说元素被删除了), 则返回true, 否则返回false
int size()	返回调用类集中元素的个数
Object[] toArray()	返回一个数组, 该数组包含了所有存储在调用类集中的元素。数组元素是类集元素的拷贝
Object[] toArray(Object array[])	返回一个数组, 该数组仅仅包含了那些类型与数组元素类型匹配的类集元素。数组元素是类集元素的拷贝。如果array的大小与匹配元素的个数相等, 它们被返回到array。如果array的大小比匹配元素的个数小, 将分配并返回一个所需大小的新数组。如果array的大小比匹配元素的个数大, 在数组中, 在类集元素之后的单元被置为null。如果任一类集元素的类型都不是array的子类型, 则引发一个ArrayStoreException异常

调用 add() 方法可以将对象加入类集。注意 add() 带一个 Object 类型的参数。因为 Object 是所有类的超类, 所以任何类型的对象可以被存储在一个类集中。然而原始类型可能不行。例如, 一个类集不能直接存储 int、char、double 等类型的值。可以通过调用 addAll() 方法将一个类集的

全部内容增加到另一个类集中。

可以通过调用 `remove()` 方法将一个对象删除。为了删除一组对象，可以调用 `removeAll()` 方法。调用 `retainAll()` 方法可以将除了一组指定的元素之外的所有元素删除。为了清空类集，可以调用 `clear()` 方法。

通过调用 `contains()` 方法，可以确定一个类集是否包含了一个指定的对象。为了确定一个类集是否包含了另一个类集的全部元素，可以调用 `containsAll()` 方法。当一个类集是空的时候，可以通过调用 `isEmpty()` 方法来予以确认。调用 `size()` 方法可以获得类集中当前元素的个数。

`toArray()` 方法返回一个数组，这个数组包含了存储在调用类集中的元素。这个方法比它看上去的能力要更重要。经常使用类数组语法来处理类集的内容是有优势的。通过在类集和数组之间提供一条路径，可以充分利用这两者的优点。

调用 `equals()` 方法可以比较两个类集是否相等。“相等”的精确含义可以不同于从类集到类集。例如，可以执行 `equals()` 方法以便用于比较存储在类集中的元素的值，换句话说，`equals()` 方法能比较对象元素的引用。

一个更加重要的方法是 `iterator()`，该方法对类集返回一个迭代程序。当使用一个类集框架时，迭代程序对于编程的成功与否是至关重要的。

16.1.2 List 接口

List 接口扩展了 Collection 并声明存储一系列元素的类集的特性。使用一个基于零的下标，元素可以通过它们在列表中的位置被插入和访问。一个列表可以包含复制元素。除了由 Collection 定义的方法之外，List 还定义了一些它自己的方法，这些方法总结在下表中。需要再次注意当类集不能被修改时，其中的几种方法将引发 `UnsupportedOperationException` 异常。当一个对象与另一个不兼容，例如企图将一个不兼容的对象加入一个类集中时，将产生 `ClassCastException` 异常。

方法	描述
<code>void add(int index, Object obj)</code>	将obj插入调用列表，插入位置的下标由index传递。任何已存在的，在插入点以及插入点之后的元素将前移。因此，没有元素被覆写
<code>Boolean addAll(int index, Collection c)</code>	将c中的所有元素插入到调用列表中，插入点的下标由index传递。在插入点以及插入点之后的元素将前移。因此，没有元素被复写。如果调用列表改变了，则返回true；否则返回false
<code>Object get(int index)</code>	返回存储在调用类集内指定下标处的对象
<code>int indexOf(Object obj)</code>	返回调用列表中obj的第一个实例的下标。如果obj不是列表中的元素，则返回-1
<code>int lastIndexOf(Object obj)</code>	返回调用列表中obj的最后一个实例的下标。如果obj不是列表中的元素，则返回-1
<code>ListIterator listIterator()</code>	返回调用列表开始的迭代程序
<code>ListIterator listIterator(int index)</code>	返回调用列表在指定下标处开始的迭代程序
<code>Object remove(int index)</code>	删除调用列表中index位置的元素并返回删除的元素。删除后，列表被压缩。也就是说，被删除元素后面的元素的下标减一
<code>Object set(int index, Object obj)</code>	用obj对调用列表内由index指定的位置进行赋值
<code>List subList(int start, int end)</code>	返回一个列表，该列表包括了调用列表中从start到end-1的元素。返回列表中的元素也被调用对象引用

对于由 Collection 定义的 `add()` 和 `addAll()` 方法，List 增加了方法 `add(int, Object)` 和 `addAll(int, Collection)`，这些方法在指定的下标处插入元素。由 Collection 定义的 `add(Object)` 和

`addAll(Collection)`的语义也被 `List` 改变了，以便它们在列表的尾部增加元素。

为了获得在指定位置存储的对象，可以用对象的下标调用 `get()` 方法。为了给类集中的一个元素赋值，可以调用 `set()` 方法，指定被改变的对象的下标。调用 `indexOf()` 或 `lastIndexOf()` 可以得到一个对象的下标。通过调用 `subList()` 方法，可以获得列表的一个指定了开始下标和结束下标的子列表。

16.1.3 集合接口

集合接口定义了一个集合，它扩展了 `Collection` 并说明了不允许复制元素的类集的特性。因此，如果试图将复制元素加到集合中时，`add()` 方法将返回 `false`。它本身并没有定义任何附加的方法。

16.1.4 SortedSet 接口

`SortedSet` 接口扩展了 `Set` 并说明了按升序排列的集合的特性。除了那些由 `Set` 定义的方法之外，由 `SortedSet` 接口说明的方法列在下表中。当没有项包含在调用集合中时，其中的几种方法会引发 `NoSuchElementException` 异常。当对象与调用集合中的元素不兼容时，将引发 `ClassCastException` 异常。如果试图使用 `null` 对象，而集合不允许 `null` 时，会引发 `NullPointerException` 异常。

方法	描述
<code>Comparator comparator()</code>	返回调用被排序集合的比较方法，如果对该集合使用自然顺序，则返回 <code>null</code>
<code>Object first()</code>	返回调用被排序集合的第1个元素
<code>SortedSet headSet(Object end)</code>	返回一个包含那些小于 <code>end</code> 的元素的 <code>SortedSet</code> ，那些元素包含在调用被排序集合中。返回被排序集合中的元素也被调用被排序集合所引用
<code>Object last()</code>	返回调用被排序集合的最后一个元素
<code>SortedSet subSet(Object start, Object end)</code>	返回一个 <code>SortedSet</code> ，它包括了从 <code>start</code> 到 <code>end-1</code> 的元素。返回类集中的元素也被调用对象所引用
<code>SortedSet tailSet(Object start)</code>	返回一个 <code>SortedSet</code> ，它包含了那些包含在分类集合中的大于等于 <code>start</code> 的元素。返回集合中的元素也被调用对象所引用

`SortedSet` 定义了几种方法，使得对集合的处理更加方便。调用 `first()` 方法，可以获得集合中的第 1 个对象。调用 `last()` 方法，可以获得集合中的最后一个元素。调用 `subSet()` 方法，可以获得排序集合的一个指定了第 1 个和最后一个对象的子集合。如果需要得到从集合的第 1 个元素开始的一个子集合，可以使用 `headSet()` 方法。如果需要获得集合尾部的一个子集合，可以使用 `tailSet()` 方法。

16.2 Collection 接口

▶ 本节视频教学录像：36 分钟

现在，读者已经熟悉了类集接口，下面开始讨论实现它们的标准类。一些类提供了完整的可被使用的工具。另一些类是抽象的，提供主框架工具，作为创建具体类集的起始点。没有一个 `Collection` 接口是同步的，就像在本章后面看到的那样，有可能获得同步版本。标准的 `Collection`

实现类总结如表所示。

类	描述
AbstractCollection	实现大多数Collection接口
AbstractList	扩展AbstractCollection并实现大多数List接口
AbstractSequentialList	为了被类集使用而扩展AbstractList, 该类集是连续而不是用随机方式访问其元素
LinkedList	通过扩展AbstractSequentialList来实现链接表
ArrayList	通过扩展AbstractList来实现动态数组
AbstractSet	扩展AbstractCollection并实现大多数Set接口
HashSet	为了使用散列表而扩展AbstractSet
TreeSet	实现存储在树中的一个集合, 扩展AbstractSet



提示：除了 Collection 接口之外，还有几个从以前版本遗留下来的类，如 Vector、Stack 和 Hashtable 等均被重新设计成支持类集的形式。这些内容将在本章后面讨论。下面讨论具体的 Collection 接口，并举例说明它们的用法。

16.2.1 ArrayList 类

ArrayList 类扩展 AbstractList 并执行 List 接口。ArrayList 支持可随需要而增长的动态数组。在 Java 中，标准数组是定长的。在数组创建之后，它们不能被加长或缩短，这也就意味着开发者必须事先知道数组可以容纳多少元素。但是在一般情况下，只有在运行时才能知道需要多大的数组。为了解决这个问题，类集框架定义了 ArrayList。本质上，ArrayList 是对象引用的一个变长数组。也就是说，ArrayList 能够动态地增加或减小其大小。数组列表以一个原始大小被创建。当超过了它的大小时，类集就会自动增大。当有对象被删除，数组就可以缩小。注意：动态数组也被从以前版本遗留下来的类 Vector 所支持。关于这一点将在后面介绍。

ArrayList 有如下的构造方法。

```
ArrayList()
ArrayList(Collection c)
ArrayList(int capacity)
```

其中第 1 个构造方法建立一个空的数组列表。第 2 个构造方法建立一个数组列表，该数组列表由类 c 中的元素初始化。第 3 个构造方法建立一个数组列表，该数组有指定的初始容量（capacity），容量是用于存储元素的基本数组的大小。当元素被追加到数组列表上时，容量会自动增加。

下面的程序是 ArrayList 的一个简单应用。首先创建一个数组列表，接着添加 String 类型的对象（回想一个引用字符串被转换成一个字符串（String）对象的方法），接着列表被显示出来。将其中的一些元素删除后，则再一次显示列表。

【范例 16-1】 ArrayList 类使用范例 1（代码 16-1.java）。

```
01 import java.util.*;
02 public class ArrayListDemo
```

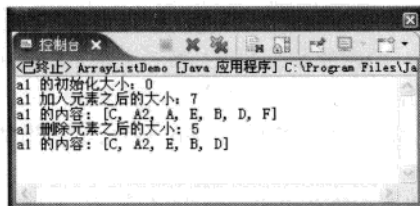
```

03 {
04     public static void main(String args[])
05     {
06         // 创建一个 ArrayList 对象
07         ArrayList al = new ArrayList();
08         System.out.println("a1 的初始化大小: " + al.size());
09         // 向 ArrayList 对象中添加新内容
10         al.add("C"); // 0 位置
11         al.add("A"); // 1 位置
12         al.add("E"); // 2 位置
13         al.add("B"); // 3 位置
14         al.add("D"); // 4 位置
15         al.add("F"); // 5 位置
16         // 把 A2 加在 ArrayList 对象的第 2 个位置
17         al.add(1, "A2"); // 加入之后的内容: CA2AEBDF
18         System.out.println("a1 加入元素之后的大小: " + al.size());
19         // 显示 ArrayList 数据
20         System.out.println("a1 的内容: " + al);
21         // 从 ArrayList 中移除数据
22         al.remove("F");
23         al.remove(2); // CA2EB D
24         System.out.println("a1 删除元素之后的大小: " + al.size());
25         System.out.println("a1 的内容: " + al);
26     }
27 }

```

【运行结果】

保存并运行程序，结果如图所示。



提示：al 开始时是空的，当添加元素后，它的大小增加了。当有元素被删除，它的大小又会变小。

【范例分析】

在前面的例子中，使用由 toString() 方法提供的默认转换显示类集的内容，toString() 方法

是从 `AbstractCollection` 继承下来的。它对简短的程序来说是足够了，但很少使用这种方法去显示实际中的类集的内容。通常编程者会提供自己的输出程序。但在下面的几个例子中，仍将采用由 `toString()` 方法创建的默认输出。

尽管当对象被存储在 `ArrayList` 对象中时，其容量会自动增加。然而，也可以通过调用 `ensureCapacity()` 方法来人工地增加 `ArrayList` 的容量。如果事先知道将在当前能够容纳的类集中存储许许多多的内容时，读者可能会想这样做。在开始时，通过一次性地增加它的容量，就能避免后面的再分配。因为再分配是很花时间的，避免不必要的处理可以提高性能。

【范例 16-2】 `ArrayList` 类使用范例 2（代码 16-2. java）。

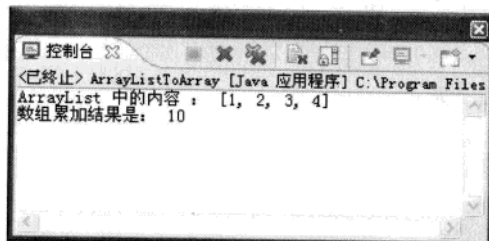
```

01  import java.util.*;
02  public class ArrayListToArray
03  {
04      public static void main(String args[])
05      {
06          // 创建一 ArrayList 对象 al
07          ArrayList al = new ArrayList();
08          // 向 ArrayList 中加入对象
09          al.add(new Integer(1));
10          al.add(new Integer(2));
11          al.add(new Integer(3));
12          al.add(new Integer(4));
13          System.out.println("ArrayList 中的内容 : " + al);
14          // 得到对象数组
15          Object ia[] = al.toArray();
16          int sum = 0;
17          // 计算数组内容
18          for (int i = 0; i < ia.length; i++)
19              sum += ((Integer) ia[i]).intValue();
20          System.out.println("数组累加结果是: " + sum);
21      }
22  }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

程序开始时创建一个整数的类集。由于不能将原始类型存储在类集中，因此类型 `Integer` 的对象被创建并被保存。接下来，`toArray()` 方法被调用，它获得了一个 `Object` 对象数组，这个数组的内容被置成整型 (`Integer`)，接下来对这些值进行求和操作。

16.2.2 LinkedList 类

`LinkedList` 类扩展了 `AbstractSequentialList` 类并实现 `List` 接口。它提供了一个链接列表的数据结构。它具有如下的两个构造方法。

```
LinkedList()
LinkedList(Collection c)
```

第 1 个构造方法建立一个空的链接列表。第 2 个构造方法建立一个链接列表，该链接列表由类 `c` 中的元素初始化。

除了它继承的方法之外，`LinkedList` 类本身还定义了一些有用的方法，这些方法主要用于操作和访问列表。使用 `addFirst()` 方法可以在列表头增加元素，使用 `addLast()` 方法可以在列表的尾部增加元素。它们的形式如下所示。

```
void addFirst(Object obj)
void addLast(Object obj)
```

在这里，`obj` 是被增加的项。

调用 `getFirst()` 方法可以获得第 1 个元素，调用 `getLast()` 方法可以得到最后一个元素。它们的形式如下所示。

```
Object getFirst()
Object getLast()
```

为了删除第 1 个元素，可以使用 `removeFirst()` 方法。为了删除最后一个元素，可以调用 `removeLast()` 方法。它们的形式如下所示。

```
Object removeFirst()
Object removeLast()
```

下面的程序是对几个 `LinkedList` 支持的方法的说明。

【范例 16-3】 `LinkedList` 类的使用（代码 16-3.java）。

```
01 import java.util.*;
02 public class LinkedListDemo
03 {
04     public static void main(String args[])
05     {
```



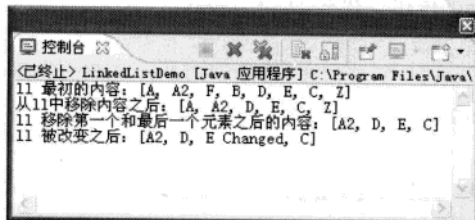
```

06      // 创建 LinkedList 对象
07      LinkedList ll = new LinkedList();
08      // 加入元素到 LinkedList 中
09      ll.add("F");
10      ll.add("B");
11      ll.add("D");
12      ll.add("E");
13      ll.add("C");
14      // 在链表的最后一个位置加上数据
15      ll.addLast("Z");
16      // 在链表的第 1 个位置上加入数据
17      ll.addFirst("A");
18      // 在链表的第 2 个元素的位置上加入数据
19      ll.add(1, "A2");
20      System.out.println("ll 最初的内容: " + ll);
21      // 从 linkedlist 中移除元素
22      ll.remove("F");
23      ll.remove(2);
24      System.out.println("从 ll 中移除内容之后: " + ll);
25      // 移除第 1 个和最后一个元素
26      ll.removeFirst();
27      ll.removeLast();
28      System.out.println("ll 移除第一个和最后一个元素之后的内容: " + ll);
29      // 取得并设置值
30      Object val = ll.get(2);
31      ll.set(2, (String) val + " Changed");
32      System.out.println("ll 被改变之后: " + ll);
33  }
34  }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

因为 LinkedList 实现 List 接口，因此调用 add(Object)将项目追加到列表的尾部，就如同

addLast()方法所做的那样。使用 add()方法的 add(int, Object)形式, 插入项目到指定的位置, 如范例程序中调用 add(1, "A2") 的举例。



提示: 通过调用 get() 和 set() 方法而使得第三个元素发生了改变。为了获得一个元素的当前值, 通过 get() 方法传递存储该元素的下标值。为了对这个下标位置赋一个新值, 通过 set() 方法传递下标和对应的新值。

16.2.3 HashSet 类

HashSet 扩展 AbstractSet 并且实现 Set 接口。它创建一个类集, 该类集使用散列表进行存储, 而散列表则通过使用称之为散列法的机制来存储信息。

在散列 (hashing) 中, 一个关键字的信息内容被用来确定唯一的一个值, 称为散列码 (hash code), 而散列码则被用来当做与关键字相连的数据的存储下标。关键字到其散列码的转换是自动执行的——看不到散列码本身。程序代码也不能直接索引散列表。散列法的优点在于即使对于大的集合, 它允许一些基本操作, 如 add()、contains()、remove() 和 size() 等方法的运行时间保持不变。

下面的构造方法定义为

```
HashSet()
HashSet(Collection c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)
```

第 1 种形式构造一个默认的散列集合, 第 2 种形式用 c 中的元素初始化散列集合, 第 3 种形式用 capacity 初始化散列集合的容量, 第 4 种形式用它的参数初始化散列集合的容量和填充比 (也称为加载容量)。填充比必须介于 0.0 与 1.0 之间, 它决定在散列集合向上调整大小之前, 有多少能被充满。具体来说, 就是当元素的个数大于散列集合容量乘以它的填充比时, 散列集合被扩大。对于没有获得填充比的构造方法, 默认使用 0.75。

HashSet 没有定义任何超类和接口提供的其他方法。

重要的是, 注意散列集合并不能确定其元素的排列顺序, 因为散列法的处理通常不让自己参与创建排序集合。如果需要排序存储, 另一种类集——TreeSet 将是一个更好的选择。

【范例 16-4】 HashSet 类的使用 (代码 16-4.java)。

```
01 import java.util.*;
02 public class HashSetDemo
03 {
04     public static void main(String args[])
05     {
06         // 创建 HashSet 对象
07         HashSet hs = new HashSet();
```

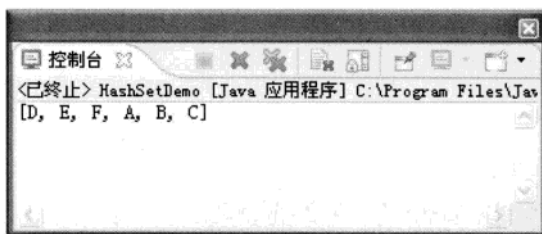
```

08      // 加入元素到 HashSet 中
09      hs.add("B");
10      hs.add("A");
11      hs.add("D");
12      hs.add("E");
13      hs.add("C");
14      hs.add("F");
15      System.out.println(hs);
16  }
17  }

```

【运行结果】

保存并运行程序，结果如图所示。



如上面解释的那样，元素并没有按顺序进行存储。

16.2.4 TreeSet 类

TreeSet 为使用树来进行存储的 Set 接口提供了一个工具，对象按升序存储。访问和检索是很快。在存储了大量的需要进行快速检索的排序信息的情况下，TreeSet 是一个很好的选择。

下面的构造方法定义为

```

TreeSet()
TreeSet(Collection c)
TreeSet(Comparator comp)
TreeSet(SortedSet ss)

```

第 1 种形式构造一个空的树集合，该树集合将根据其元素的自然顺序按升序排序。第 2 种形式构造一个包含了 c 的元素的树集合。第 3 种形式构造一个空的树集合，它按照由 comp 指定的比较方法进行排序（比较方法将在本章后面介绍）。第 4 种形式构造一个包含了 ss 的元素的树集合。下面是一个 TreeSet 的使用范例。

【范例 16-5】 TreeSet 的使用（代码 16-5. java）。

```

01  import java.util.*;
02  public class TreeSetDemo

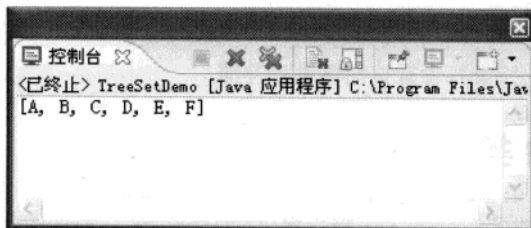
```

```

03  {
04      public static void main(String args[])
05      {
06          // 创建一 TreeSet 对象
07          TreeSet ts = new TreeSet();
08          // 加入元素到 TreeSet 中
09          ts.add("C");
10          ts.add("A");
11          ts.add("B");
12          ts.add("E");
13          ts.add("F");
14          ts.add("D");
15          System.out.println(ts);
16      }
17  }
    
```

【运行结果】

保存并运行程序，结果如图所示。



正如上面解释的那样，因为 TreeSet 按树存储其元素，因此它们被按照排序次序自动排序。

16.3 通过迭代方法访问类集

通常开发者希望通过循环输出类集中的元素。例如，可能会希望显示每一个元素。到目前为止，处理这个问题最简单的方法是使用 `iterator()`，`iterator()` 是一个或者实现 `Iterator`，或者实现 `ListIterator` 接口的对象。`Iterator` 可以完成通过循环输出类集内容，从而获得或删除元素。`ListIterator` 扩展了 `Iterator`，允许双向遍历列表，并且可以修改单元。`Iterator` 接口说明的方法总结在下表中。

方法	描述
<code>boolean hasNext()</code>	如果存在更多的元素，则返回 <code>true</code> ，否则返回 <code>false</code>
<code>Object next()</code>	返回下一个元素。如果没有下一个元素，则引发 <code>NoSuchElementException</code> 异常
<code>void remove()</code>	删除当前元素，如果试图在调用 <code>next()</code> 方法之后调用 <code>remove()</code> 方法，则引发 <code>IllegalStateException</code> 异常

ListIterator 接口说明的方法总结在下表中。

方法	描述
void add(Object obj)	将obj插入列表中的一个元素之前, 该元素在下次调用next()方法时被返回
boolean hasNext()	如果存在下一个元素, 则返回true, 否则返回false
boolean hasPrevious()	如果存在前一个元素, 则返回true, 否则返回false
Object next()	返回下一个元素, 如果不存在下一个元素, 则引发一个NoSuchElementException异常
int nextIndex()	返回下一个元素的下标, 如果不存在下一个元素, 则返回列表的大小
Object previous()	返回前一个元素, 如果前一个元素不存在, 则引发一个NoSuchElementException异常
int previousIndex()	返回前一个元素的下标, 如果前一个元素不存在, 则返回-1
void remove()	从列表中删除当前元素。如果remove()方法在next()方法或previous()方法调用之前被调用, 则引发一个IllegalStateException异常
void set(Object obj)	将obj赋给当前元素。这是上一次调用next()方法或previous()方法最后返回的元素

下面介绍使用迭代的方法。

在通过迭代方法访问类集之前, 必须得到一个迭代方法。每一个 Collection 类都提供一个 iterator() 方法, 该方法返回一个对类集的迭代方法。通过使用这个迭代方法对象, 可以一次一个地访问类集中的每一个元素。通常, 使用迭代方法通过循环输出类集的内容, 具体的操作步骤如下。

- ① 通过调用类集的 iterator() 方法获得对类集的迭代方法。
- ② 建立一个调用 hasNext() 方法的循环, 只要 hasNext() 返回 true, 就进行循环迭代。
- ③ 在循环内部, 通过调用 next() 方法来得到每一个元素。

对于执行 List 的类集, 也可以通过调用 ListIterator 来获得迭代方法。正如上面解释的那样, 列表迭代方法提供了前向或后向访问类集的能力, 并且可以修改元素, 否则 ListIterator 如同 Iterator 功能一样。程序 IteratorDemo.java 是一个实现上述描述的例子, 说明了 Iterator 和 ListIterator 的使用方法。在范例中使用的是 ArrayList 类, 但是原则上它是适用于任何类型的类集的。当然, ListIterator 只适用于那些实现 List 接口的类集。

【范例 16-6】 通过迭代方法访问类集 (代码 16-6. java)。

```

01  import java.util.*;
02  public class IteratorDemo
03  {
04      public static void main(String args[])
05      {
06          // 创建一个 ArrayList 数组
07          ArrayList al = new ArrayList();
08          // 加入元素到 ArrayList 中
09          al.add("C");
10          al.add("A");
11          al.add("E");

```



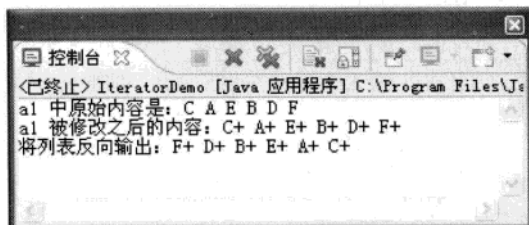
```

12     al.add("B");
13     al.add("D");
14     al.add("F");
15     // 使用 iterator 显示 a1 中的内容
16     System.out.print("a1 中原始内容是: ");
17     Iterator itr = al.iterator();
18     while (itr.hasNext())
19     {
20         Object element = itr.next();
21         System.out.print(element + " ");
22     }
23     System.out.println();
24     // 在 ListIterator 中修改内容
25     ListIterator litr = al.listIterator();
26     while (litr.hasNext())
27     {
28         Object element = litr.next();
29         // 用 set 方法修改其内容
30         litr.set(element + "+");
31     }
32     System.out.print("a1 被修改之后的内容: ");
33     itr = al.iterator();
34     while (itr.hasNext())
35     {
36         Object element = itr.next();
37         System.out.print(element + " ");
38     }
39     System.out.println();
40     // 下面是将列表中的内容反向输出
41     System.out.print("将列表反向输出: ");
42     // hasPrevious 由后向前输出
43     while (litr.hasPrevious())
44     {
45         Object element = litr.previous();
46         System.out.print(element + " ");
47     }
48     System.out.println();
49 }
50 }

```


【运行结果】

保存并运行程序，结果如图所示。



值得注意的是列表是被反向显示的。在列表被修改之后，`litr` 指向列表的末端（记住，当到达列表末端时，`litr.hasNext()`方法返回 `false`）。为了反向遍历列表，程序继续使用 `litr`，但这一次，程序将检测它是否有前一个元素，只要它有前一个元素，该元素就被获得并被显示出来。

16.4 处理映射

本节视频教学录像：38 分钟

Java 2 还在 `java.util` 中增加了映射。映射（map）是一个存储关键字和值的关联，或者说是“关键字/值”对的对象，即给定一个关键字，可以得到它的值。关键字和值都是对象，关键字必须是唯一的，但值是可以被复制的。有的映射可以接收 `null` 关键字和 `null` 值，有的则不能。

16.4.1 映射接口

正因为映射接口定义了映射的特征和本质，所以从这里开始讨论映射。表中所列为支持映射的接口。

接口	描述
<code>Map</code>	映射唯一关键字到值
<code>Map.Entry</code>	描述映射中的元素（“关键字/值”对）。是 <code>Map</code> 的一个内部类
<code>SortedMap</code>	扩展 <code>Map</code> 以便关键字按升序保持

下面对每个接口依次进行讨论。

1. Map 接口

`Map` 接口映射唯一关键字到值。关键字（key）是以后用于检索值的对象。给定一个关键字和一个值，可以存储这个值到一个 `Map` 对象中。当这个值被存储以后，就可以使用它的关键字来检索它。由 `Map` 说明的方法总结在下表中。当调用的映射中没有项存在时，其中的几种方法会引发一个 `NoSuchElementException` 异常。而当对象与映射中的元素不兼容时，则会引发一个 `ClassCastException` 异常。如果试图使用映射不允许使用的 `null` 对象，则会引发一个 `NullPointerException` 异常。当试图改变一个不允许修改的映射时，则会引发一个 `UnsupportedOperationException` 异常。

方法	描述
<code>void clear()</code>	从调用映射中删除所有的关键字/值对
<code>boolean containsKey(Object k)</code>	如果调用映射中包含了作为关键字的 <code>k</code> ，则返回 <code>true</code> ，否则返回 <code>false</code>

续表

方法	描述
boolean containsValue(Object v)	如果映射中包含了作为值的 v, 则返回 true, 否则返回 false
Set entrySet()	返回包含了映射中的项的集合 (Set)。该集合包含了类型 Map.Entry 的对象。这个方法为调用映射提供了一个集合“视图”
Boolean equals(Object obj)	如果 obj 是一个 Map 并包含相同的输入, 则返回 true, 否则返回 false
Object get(Object k)	返回与关键字 k 相关联的值
int hashCode()	返回调用映射的散列码
boolean isEmpty()	如果调用映射是空的, 则返回 true, 否则返回 false
Set keySet()	返回一个包含调用映射中关键字的集合 (Set)。这个方法为调用映射的关键字提供了一个集合“视图”
Object put(Object k, Object v)	将一个输入加入调用映射, 改写原先与该关键字相关联的值。关键字和值分别为 k 和 v。如果关键字已经不存在了, 则返回 null; 否则, 返回原先与关键字相关联的值
void putAll(Map m)	将所有来自 m 的输入加入调用映射
Object remove(Object k)	删除关键字等于 k 的输入
int size()	返回映射中关键字/值对的个数
Collection values()	返回一个包含了映射中的值的类集。这个方法为映射中的值提供了一个类集“视图”映射循环使用两个基本操作: get() 和 put()。使用 put() 方法可以将一个指定了关键字和值的值加入映射。为了得到值, 可以通过将关键字作为参数来调用 get() 方法, 调用返回该值

正如前面谈到的, 映射不是类集, 但可以获得映射的类集“视图”。为了实现这种功能, 可以使用 entrySet() 方法, 它返回一个包含了映射中元素的集合 (Set)。为了得到关键字的类集“视图”, 可以使用 keySet() 方法。为了得到值的类集“视图”, 可以使用 values() 方法。类集“视图”是将映射集成到类集框架内的手段。

2. SortedMap 接口

SortedMap 接口扩展了 Map, 它确保了各项按关键字升序排序。由 SortedMap 说明的方法总结在下表中。当调用映射中没有的项时, 其中的几种方法将引发一个 NoSuchElementException 异常。当对象与映射中的元素不兼容时, 则会引发一个 ClassCastException 异常。当试图使用映射不允许使用的 null 对象时, 则会引发一个 NullPointerException 异常。

方法	描述
Comparator comparator()	返回调用排序映射的比较方法。如果调用映射使用的是自然顺序的话, 则返回 null
Object firstKey()	返回调用映射的第 1 个关键字
SortedMap headMap(Object end)	返回一个排序映射, 该映射包含了那些关键字小于 end 的映射输入
Object lastKey()	返回调用映射的最后一个关键字
SortedMap subMap(Object start, Object end)	返回一个映射, 该映射包含了那些关键字大于等于 start 同时小于 end 的输入
SortedMap tailMap(Object start)	返回一个映射, 该映射包含了那些关键字大于等于 start 的输入排序映射允许对子映射 (换句话说, 就是映射的子集) 进行高效的处理。使用 headMap()、tailMap() 或 subMap() 方法可以获得子映射。调用 firstKey() 方法可以获得集合的第 1 个关键字, 而调用 lastKey() 方法则可获得集合的最后一个关键字

3. Map.Entry 接口

Map.Entry 接口使得可以操作映射的输入。回想由 Map 接口说明的 entrySet() 方法, 调用该方法可返回一个包含映射输入的集合 (Set), 这些集合元素的每一个都是一个 Map.Entry 对象。

下表总结了由该接口说明的方法。

方法	描述
<code>boolean equals(Object obj)</code>	如果 <code>obj</code> 是一个关键字和值都与调用对象相等的 <code>Map.Entry</code> ，则返回 <code>true</code>
<code>Object getKey()</code>	返回该映射项的关键字
<code>Object getValue()</code>	返回该映射项的值
<code>int hashCode()</code>	返回该映射项的散列值
<code>Object setValue(Object v)</code>	将这个映射输入的值赋给 <code>v</code> 。如果 <code>v</code> 不是映射的正确类型，则引发一个 <code>ClassCastException</code> 异常。如果 <code>v</code> 存在问题，则引发一个 <code>IllegalArgumentException</code> 异常。如果 <code>v</code> 是 <code>null</code> ，而映射又不允许是 <code>null</code> 关键字，则引发一个 <code>NullPointerException</code> 异常。如果映射不能被改变，则会引发一个 <code>UnsupportedOperationException</code> 异常

16.4.2 映射类

有几个类提供了映射接口的实现。可以被用做映射的类如表所示。

类	描述
<code>AbstractMap</code>	实现大多数的 <code>Map</code> 接口
<code>HashMap</code>	将 <code>AbstractMap</code> 扩展到使用散列表
<code>TreeMap</code>	将 <code>AbstractMap</code> 扩展到使用树

注意 `AbstractMap` 对 3 个具体的映射实现来说，是一个超类。`WeakHashMap` 实现一个使用“弱关键字”的映射，它允许映射中的元素，当该映射的关键字不再被使用时，被放入回收站。关于这个类在这里不做更深入的讨论，其他的类将在下面介绍。

1. `HashMap` 类

`HashMap` 类使用散列表实现 `Map` 接口。这允许一些基本操作，如 `get()` 和 `put()` 的运行时间保持恒定，即便对大型的集合也是这样的。下面的构造方法定义为。

- (1) `HashMap()`
- (2) `HashMap(Map m)`
- (3) `HashMap(int capacity)`
- (4) `HashMap(int capacity, float fillRatio)`

第 1 种形式构造一个默认的散列映射。第 2 种形式用 `m` 的元素初始化散列映射。第 3 种形式将散列映射的容量初始化为 `capacity`。第 4 种形式用它的参数同时初始化散列映射的容量和填充比。容量和填充比的含义与前面介绍的 `HashSet` 中的容量和填充比相同。

`HashMap` 实现 `Map` 并扩展 `AbstractMap`。它本身并没有增加任何新的方法。应该注意的是：散列映射并不保证它的元素的顺序。因此，元素加入散列映射的顺序并不一定是它们被迭代方法读出的顺序。

下面的程序举例说明了 `HashMap`。它将名字映射到账目资产平衡表。应注意集合“视图”是如何获得和被使用的。

【范例 16-7】 将名字映射到账目资产平衡表（代码 16-7. java）。

```

01 import java.util.*;
02 public class HashMapDemo
03 {

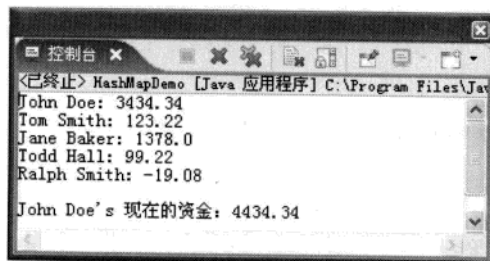
```



```
04     public static void main(String args[])
05     {
06         // 创建 HashMap 对象
07         HashMap hm = new HashMap();
08         // 加入元素到 HashMap 中
09         hm.put("John Doe", new Double(3434.34));
10         hm.put("Tom Smith", new Double(123.22));
11         hm.put("Jane Baker", new Double(1378.00));
12         hm.put("Todd Hall", new Double(99.22));
13         hm.put("Ralph Smith", new Double(-19.08));
14         // 返回包含映射中项的集合
15         Set set = hm.entrySet();
16         // 用 Iterator 得到 HashMap 中的内容
17         Iterator i = set.iterator();
18         // 显示元素
19         while (i.hasNext())
20         {
21             // Map.Entry 可以操作映射的输入
22             Map.Entry me = (Map.Entry) i.next();
23             System.out.print(me.getKey() + ": ");
24             System.out.println(me.getValue());
25         }
26         System.out.println();
27         // 让 John Doe 中的值增加 1000
28         double balance = ((Double) hm.get("John Doe")).doubleValue();
29         // 用新的值替换掉旧的值
30         hm.put("John Doe", new Double(balance + 1000));
31         System.out.println("John Doe's 现在的资金: " + hm.get("John Doe"));
32     }
33 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

程序开始创建了一个散列映射，然后将名字的映射增加到平衡表中。接下来，映射的内容通过使用由调用方法 `entrySet()` 而获得的集合“视图”而显示出来。关键字和值通过调用由 `Map.Entry` 定义的 `getKey()` 和 `getValue()` 方法而显示。注意存款是如何被制成 John Doe 的账目的。`put()` 方法自动用新值替换与指定关键字相关联的原先的值。因此，在 John Doe 的账目被更新后，散列映射则仍然仅仅保留一个“John Doe”账目。

2. TreeMap 类

`TreeMap` 类通过使用树实现 `Map` 接口。`TreeMap` 提供了按排序顺序存储关键字/值对的有效手段，同时允许快速检索。应该注意的是，不像散列映射，树映射保证它的元素按照关键字升序排序。

下面的 `TreeMap` 构造方法定义为。

- (1) `TreeMap()`
- (2) `TreeMap(Comparator comp)`
- (3) `TreeMap(Map m)`
- (4) `TreeMap(SortedMap sm)`

第 1 种形式构造一个空树的映射，该映射使用其关键字的自然顺序来排序。第 2 种形式构造一个空的基于树的映射，该映射通过使用 `Comparator comp` 来排序（比较方法 `Comparator` 将在本章后面讨论）。第 3 种形式用从 `m` 的输入初始化树映射，该映射使用关键字的自然顺序来排序。第 4 种形式用从 `sm` 的输入来初始化一个树映射，该映射将按与 `sm` 相同的顺序来排序。

`TreeMap` 实现 `SortedMap` 并且扩展 `AbstractMap`。而它本身并没有另外定义其他的方法。

下面的程序重新使用前面的例子运行，以便在其中使用 `TreeMap`。

【范例 16-8】 `TreeMap` 的使用（代码 16-8. java）。

```

01  import java.util.*;
02  public class TreeMapDemo
03  {
04      public static void main(String args[])
05      {
06          // 创建 TreeMap 对象
07          TreeMap tm = new TreeMap();
08          // 加入元素到 TreeMap 中
09          tm.put(new Integer(10000 - 2000), "张三");
10          tm.put(new Integer(10000 - 1500), "李四");
11          tm.put(new Integer(10000 - 2500), "王五");
12          tm.put(new Integer(10000 - 5000), "赵六");
13          Collection col = tm.values();
14          Iterator i = col.iterator();
15          System.out.println("按工资由高到低顺序输出：");
16          while (i.hasNext())
17          {

```



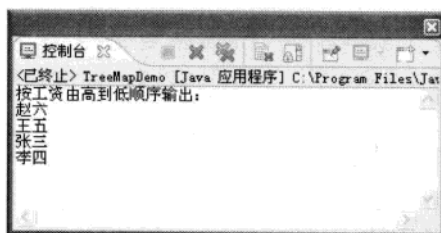
```

18         System.out.println(i.next());
19     }
20 }
21 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

注意对关键字进行了排序。然而在这种情况下，它们用名字而不是用姓进行了排序。可以通过在创建映射时，指定一个比较方法来改变这种排序。下一小节将介绍如何做。

16.4.3 比较方法

TreeSet 和 TreeMap 都按排序顺序存储元素。然而，精确定义采用何种“排序顺序”则是比较方法。通常在默认的情况下，这些类通过使用被 Java 称之为“自然顺序”的顺序存储它们的元素，而这种顺序通常也是你所需要的（A 在 B 的前面，1 在 2 的前面，等等）。如果需要用不同的方法对元素进行排序，可以在构造集合或映射时，指定一个 Comparator 对象。这样做为开发者提供了一种精确控制如何将元素储存在排序类集和映射中的能力。

Comparator 接口定义了两个方法：compare() 和 equals()。这里给出的 compare() 方法按顺序比较了两个元素。

```
int compare(Object obj1, Object obj2)
```

obj1 和 obj2 是被比较的两个对象。当两个对象相等时，该方法返回 0；当 obj1 大于 obj2 时，返回一个正值；否则，返回一个负值。如果用于比较的对象的类型不兼容的话，该方法会引发一个 ClassCastException 异常。通过改写 compare()，可以改变对象排序的方式。例如，通过创建一个颠倒比较输出的比较方法，可以实现按逆向排序。

这里给出的 equals() 方法，用于测试一个对象是否与调用比较方法相等。

```
boolean equals(Object obj)
```

obj 是被用来进行相等测试的对象。如果 obj 和调用对象都是 Comparator 的对象，并且使用相同的排序，该方法则返回 true，否则返回 false。重载 equals() 方法是没有必要的，大多数简单的比较方法都不这样做。

下面是一个说明定制的比较方法能力的例子。该例子实现 compare() 方法以便它按照正常顺

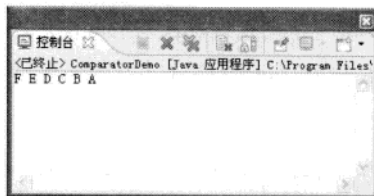
序的逆向进行操作。因此，它使得一个树集合按逆向的顺序进行存储。

【范例 16-9】 定制的比较方法能力（代码 16-9. java）。

```
01  import java.util.*;
02  class MyComp implements Comparator
03  {
04      public int compare(Object o1, Object o2)
05      {
06          String aStr, bStr;
07          aStr = (String)o1;
08          bStr = (String)o2;
09          return bStr.compareTo(aStr);
10      }
11  }
12  public class ComparatorDemo
13  {
14      public static void main(String args[])
15      {
16          // 创建一个 TreeSet 对象
17          TreeSet ts = new TreeSet(new MyComp());
18          // 向 TreeSet 对象中加入内容
19          ts.add("C");
20          ts.add("A");
21          ts.add("B");
22          ts.add("E");
23          ts.add("F");
24          ts.add("D");
25          // 得到 Iterator 的实例化对象
26          Iterator i = ts.iterator();
27          // 显示全部内容
28          while (i.hasNext())
29          {
30              Object element = i.next();
31              System.out.print(element + " ");
32          }
33      }
34  }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

仔细观察实现 `Comparator` 并覆写 `compare()` 方法的 `MyComp` 类（正如前面所解释的那样，覆写 `equals()` 方法既不是必须的，也不是常用的）。在 `compare()` 方法内部，`String` 方法 `compareTo()` 比较两个字符串。然而由 `bStr`——不是 `aStr`——调用 `compareTo()` 方法，会导致比较的结果被逆向。

对应一个更实际的例子，下面是用 `TreeMap` 程序实现存储账目资产平衡表例子的程序。在前面介绍的程序中，账目是按名进行排序的，但程序是以按照名字进行排序开始的。下面的程序按姓对账目进行排序。为了实现这种功能，程序使用了比较方法来比较每一个账目下姓的排序，得到的映射是按姓进行排序的。

【范例 16-10】 使用 `TreeMap` 程序实现存储账目资产平衡表（代码 16-10. java）。

```
01 import java.util.*;
02 class Employee implements Comparator
03 {
04     public int compare(Object a, Object b)
05     {
06         int k;
07         String aStr, bStr;
08         aStr = (String) a;
09         bStr = (String) b;
10         k = aStr.compareTo(bStr);
11         if(k==0)
12             return aStr.compareTo(bStr);
13         else
14             return k;
15     }
16 }
17
18 public class TreeMapDemo2
19 {
20     public static void main(String args[])
21     {
22         // 创建一个 TreeMap 对象
23         TreeMap tm = new TreeMap(new Employee());
24         // 向 Map 对象中插入元素
```

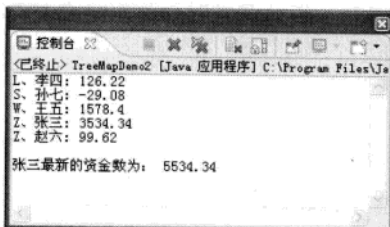
```

25         tm.put("Z、张三", new Double(3534.34));
26         tm.put("L、李四", new Double(126.22));
27         tm.put("W、王五", new Double(1578.40));
28         tm.put("Z、赵六", new Double(99.62));
29         tm.put("S、孙七", new Double(-29.08));
30         Set set = tm.entrySet();
31         Iterator itr = set.iterator();
32         while(itr.hasNext())
33         {
34             Map.Entry me = (Map.Entry)itr.next();
35             System.out.print(me.getKey() + ": ");
36             System.out.println(me.getValue());
37         }
38         System.out.println();
39         double balance = ((Double)tm.get("Z、张三")).doubleValue();
40         tm.put("Z、张三", new Double(balance + 2000));
41         System.out.println("张三最新的资金数为: " + tm.get("Z、张三"));
42     }
43 }

```

【运行结果】

保存并运行程序，结果如图所示。



16.5 从以前版本遗留下来的类和接口

java.util 的最初版本中不包括类集框架。取而代之，它定义了几个类和接口提供专门的方法用于存储对象。随着在 Java 2 中引入类集，有几种最初的类被重新设计成支持类集接口，因此它们与框架完全兼容。尽管实际上没有类被摒弃，但其中某些仍被认为是过时的。当然，在那些重复从以前版本遗留下来的类的功能性的地方，通常都愿意用类集编写新的代码程序。一般来说，对从以前版本遗留下来的类的支持是因为仍然存在着大量使用它们的基本代码，包括现在仍在被 Java 2 的应用编程接口（API）使用的程序。

另一点，没有一个类集类是同步的。但是所有的从以前版本遗留下来的类都是同步的。这一区别在有些情况下是很重要的。当然，通过使用由 Collections 提供的算法也很容易实现类集同步。

由 java.util 定义的从以前版本遗留下来的类如下：Dictionary、Hashtable、Properties、Stack 和 Vector。

有一个枚举（Enumeration）接口是从以前版本遗留下来的。下面将依次介绍 Enumeration 和每一种从以前版本遗留下来的类。

16.5.1 Enumeration 接口

Enumeration 接口定义了可以对一个对象的类集中的元素进行枚举（一次获得一个）的方法。这个接口尽管没有被摒弃，但已经被 Iterator 所替代。Enumeration 对新程序来说是过时的。然而它仍被几种从以前版本遗留下来的类（例如 Vector 和 Properties）所定义的方法使用，被几种其他的 API 类所使用，以及被目前广泛使用的应用程序所使用。

Enumeration 指定下面的两个方法。

(1) boolean hasMoreElements()

(2) Object nextElement()

执行后，当仍有更多的元素可提取时，hasMoreElements() 方法一定返回 true。当所有元素都被枚举了，则返回 false。nextElement() 方法将枚举中的下一个对象作为一个类属 Object 的引用而返回。也就是每次调用 nextElement() 方法获得枚举中的下一个对象。

16.5.2 Vector 类

Vector 实现动态数组，这与 ArrayList 相似，但两者不同的是：Vector 是同步的，并且它包含了许多不属于类集框架的从以前版本遗留下来的方法。随着 Java 2 的公布，Vector 被重新设计来扩展 AbstractList 和实现 List 接口，因此现在它与类集是完全兼容的。

下面列出 Vector 的构造方法。

Vector()

Vector(int size)

Vector(int size, int incr)

Vector(Collection c)

第 1 种形式创建一个原始大小为 10 的默认矢量。第 2 种形式创建一个其原始容量由 size 指定的矢量。第 3 种形式创建一个其原始容量由 size 指定，并且它的增量由 incr 指定的矢量。增量指定了矢量每次允许向上改变大小的元素的个数。第 4 种形式创建一个包含了类集 c 中元素的矢量。这个构造方法是在 Java 2 中新增加的。

所有的矢量开始都有一个原始的容量。在这个原始容量达到以后，下一次再试图向矢量中存储对象时，矢量会自动为那个对象分配空间，同时为别的对象增加额外的空间。通过分配超过需要的内存，矢量减小了可能产生的分配的次数。这种次数的减少是很重要的，因为分配内存是很花时间的。在每一次的再分配中，分配的额外空间的总数由在创建矢量时指定的增量来确定。如果没有指定增量，在每个分配周期，矢量的大小增加一倍。

Vector 定义了下面的保护数据成员。

int capacityIncrement;


```
int elementCount;
Object elementData[ ];
```

增量值被存储在 `capacityIncrement` 中, 矢量中的当前元素的个数被存储在 `elementCount` 中, 保存矢量的数组被存储在 `elementData` 中。

除了由 `List` 定义的类集方法之外, `Vector` 还定义了几个从以前版本遗留下来的方法, 这些方法列在下表中。

方法	描述
<code>final void addElement(Object element)</code>	将由 <code>element</code> 指定的对象加入矢量
<code>int capacity()</code>	返回矢量的容量
<code>Object clone()</code>	返回调用矢量的一个拷贝
<code>Boolean contains(Object element)</code>	如果 <code>element</code> 被包含在矢量中, 则返回 <code>true</code> ; 如果不包含于其中, 则返回 <code>false</code>
<code>void copyInto(Object array[])</code>	将包含在调用矢量中的元素复制到由 <code>array</code> 指定的数组中
<code>Object elementAt(int index)</code>	返回由 <code>index</code> 指定位置的元素
<code>Enumeration elements()</code>	返回矢量中元素的一个枚举
<code>Object firstElement()</code>	返回矢量的第 1 个元素
<code>int indexOf(Object element)</code>	返回 <code>element</code> 首次出现的位置下标。如果对象不在矢量中, 则返回 -1
<code>int indexOf(Object element, int start)</code>	返回 <code>element</code> 在矢量中, 在 <code>start</code> 及其之后第 1 次出现的位置下标。如果该对象不属于矢量的这一部分, 则返回 -1
<code>void insertElementAt(Object element, int index)</code>	在矢量中, 在由 <code>index</code> 指定的位置处加入 <code>element</code>
<code>boolean isEmpty()</code>	如果矢量是空的, 则返回 <code>true</code> ; 如果它包含了一个或多个元素, 则返回 <code>false</code>
<code>Object lastElement()</code>	返回矢量中的最后一个元素
<code>int lastIndexOf(Object element)</code>	返回 <code>element</code> 在矢量中最后一次出现的位置下标。如果对象不包含在矢量中, 则返回 -1
<code>int lastIndexOf(Object element, int start)</code>	返回 <code>element</code> 在矢量中, 在 <code>start</code> 之前最后一次出现的位置下标。如果该对象不属于矢量的这一部分, 则返回 -1
<code>void removeAllElements()</code>	清空矢量, 在这个方法执行以后, 矢量的大小为 0
<code>boolean removeElement(Object element)</code>	从矢量中删除 <code>element</code> 。对于指定的对象, 矢量中如果有其多个实例, 则其中的第 1 个实例被删除。如果成功删除, 则返回 <code>true</code> ; 如果没有发现对象, 则返回 <code>false</code>
<code>void removeElementAt(int index)</code>	删除由 <code>index</code> 指定位置处的元素
<code>void setElementAt(Object element, int index)</code>	将由 <code>index</code> 指定的位置分配给 <code>element</code>
<code>void setSize(int size)</code>	将矢量中元素的个数设为 <code>size</code> 。如果新的长度小于老的长度, 元素将丢失; 如果新的长度大于老的长度, 则在其后增加 <code>null</code> 元素
<code>int size()</code>	返回矢量中当前元素的个数
<code>String toString()</code>	返回矢量的字符串等价形式
<code>void trimToSize()</code>	将矢量的容量设为与其当前拥有的元素的个数相等

因为 `Vector` 实现 `List`, 所以可以像使用 `ArrayList` 的一个实例那样使用矢量。也可以使用它的从以前版本遗留下来的方法来操作它。例如, 在后面实例化 `Vector`, 可以通过调用 `addElement()` 方法而为其增加一个元素。调用 `elementAt()` 方法可以获得指定位置处的元素。

调用 `firstElement()` 方法可以得到矢量的第 1 个元素, 调用 `lastElement()` 方法可以检索到矢量的最后一个元素, 使用 `indexOf()` 和 `lastIndexOf()` 方法可以获得元素的下标, 调用 `removeElement()` 或 `removeElementAt()` 方法可以删除元素。

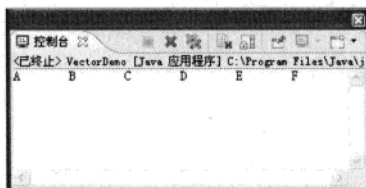
下面的程序使用矢量存储不同类型的数值对象。程序说明了几种由 Vector 定义的从以前版本遗留下来的方法，同时它也说明了枚举（Enumeration）接口。

【范例 16-11】 使用矢量存储不同类型的数值对象（代码 16-11. java）。

```
01 import java.util.*;
02 public class VectorDemo
03 {
04     public static void main(String[] args)
05     {
06         Vector v = new Vector();
07         v.add("A");
08         v.add("B");
09         v.add("C");
10         v.add("D");
11         v.add("E");
12         v.add("F");
13         Enumeration e = v.elements();
14         while(e.hasMoreElements())
15         {
16             System.out.print(e.nextElement()+"\t");
17         }
18     }
19 }
```

【运行结果】

保存并运行程序，结果如图所示。



随着 Java 2 的公布，Vector 增加了对迭代方法的支持。现在可以使用迭代方法来替代枚举去遍历对象（正如前面的程序所做的那样）。例如，下面的基于迭代方法的程序代码可以被替换到上面的程序中。

```
Iterator i = v.iterator();
while(i.hasNext())
{
    System.out.print(i.next()+"\t");
}
```

建议不要使用编写枚举新的程序代码，所以通常可以使用迭代方法来对矢量的内容进行枚举。当然，业已存在的大量的老程序采用了枚举。不过幸运的是，枚举和迭代方法的工作方式几乎相同。

16.5.3 Stack 类

Stack 是 Vector 的一个子类，它实现标准的后进先出堆栈。Stack 仅仅定义了创建空堆栈的默认构造方法。Stack 包括了由 Vector 定义的所有方法，同时增加了几种它自己定义的方法，具体总结在下表中。

方法	描述
boolean empty()	如果堆栈是空的，则返回 true；当堆栈包含元素时，则返回 false
Object peek()	返回位于栈顶的元素，但是并不在堆栈中删除它
Object pop()	返回位于栈顶的元素，并在进程中删除它
Object push(Object element)	将 element 压入堆栈，同时也返回 element
int search(Object element)	在堆栈中搜索 element，如果发现了，则返回它相对于栈顶的偏移量。否则，返回-1 调用 push()方法可将一个对象压入栈顶。调用 pop()方法可以删除和返回栈顶的元素。当调用堆栈是空的时，如果调用 pop()方法，将引发一个 EmptyStackException 异常。调用 peek()方法返回但不删除栈顶的对象。调用 empty()方法，当堆栈中没有元素时，返回 true
search()	方法确定一个对象是否存在于堆栈，并且返回将其指向栈顶所需的弹出次数。

下面是一个创建堆栈的例子，将几个整型 (Integer) 对象压入堆栈，然后再将它们弹出。

【范例 16-12】 创建堆栈 (代码 16-12. java)。

```

01  import java.util.*;
02  public class StackDemo
03  {
04      static void showpush(Stack st, int a)
05      {
06          st.push(new Integer(a));
07          System.out.println("入栈(" + a + ")");
08          System.out.println("Stack: " + st);
09      }
10
11      static void showpop(Stack st)
12      {
13          System.out.print("出栈 -> ");
14          Integer a = (Integer) st.pop();
15          System.out.println(a);
16          System.out.println("Stack: " + st);
17      }
18

```

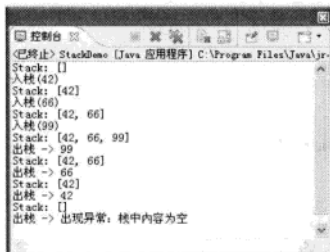
```

19     public static void main(String args[])
20     {
21         Stack st = new Stack();
22         System.out.println("Stack: " + st);
23         showpush(st, 42);
24         showpush(st, 66);
25         showpush(st, 99);
26         showpop(st);
27         showpop(st);
28         showpop(st);
29         //出栈的时候会有一个 EmptyStackException 的异常，需要进行异常处理
30         try {
31             showpop(st);
32         } catch (EmptyStackException e) {
33             System.out.println("出现异常：栈中内容为空");
34         }
35     }
36 }

```

【运行结果】

下图是由该程序产生的输出。注意对于 `EmptyStackException` 的异常处理程序是如何被捕获的，以便能够从容地处理堆栈的下溢。



16.5.4 Dictionary 类

字典 (Dictionary) 是一个表示关键字/值存储库的抽象类，同时它的操作也很像映射 (Map)。给定一个关键字和值，可以将值存储到字典 (Dictionary) 对象中。一旦这个值被存储了，就能够用它的关键字来检索它。因此与映射一样，字典可以被当做关键字/值对列表来考虑。尽管在 Java 2 中并没有摒弃字典 (Dictionary)，但由于它被映射 (Map) 所取代，从而被认为是过时的。然而由于目前 Dictionary 被广泛地使用，因此这里仍对它进行详细的讨论。

由 Dictionary 定义的抽象方法如表所示。

方法	描述
<code>Enumeration elements()</code>	返回对包含在字典中的值的枚举

续表

方法	描述
Object get(Object key)	返回一个包含与 key 相连的值的对象。如果 key 不在字典中, 则返回一个空对象
boolean isEmpty()	如果字典是空的, 则返回 true; 如果字典中至少包含一个关键字, 则返回 false
Enumeration keys()	返回包含在字典中的关键字的枚举
Object put(Object key, Object value)	将一个关键字和它的值插入字典中。如果 key 已经不在字典中了, 则返回 null; 如果 key 已经在字典中了, 则返回与 key 相关联的前一个值
Object remove(Object key)	删除 key 和它的值, 返回与 key 相关联的值。如果 key 不在字典中, 则返回 null
int size()	返回字典中的项数, 使用 put()方法在字典中增加关键字和值。使用 get()方法检索给定关键字的值。当分别使用 keys()和 elements()方法进行枚举 (Enumeration) 时, 关键字和值可以分别逐个地返回
size()	方法返回存储在字典中的关键字/值对的个数。当字典是空的时候, isEmpty()返回 true。使用 remove()方法可以删除关键字/值对



提 示: Dictionary 类是过时的。应该执行 Map 接口去获得关键字/值存储的功能。

16.5.5 Hashtable 类

散列表 (Hashtable) 是原始 java.util 中的一部分, 同时也是 Dictionary 的一个具体实现。然而, Java 2 重新设计了散列表 (Hashtable), 以便它也能实现映射 (Map) 接口。因此现在 Hashtable 也被集成到类集框架中。它与 HashMap 相似, 但它是同步的。和 HashMap 一样, Hashtable 将关键字/值对存储到散列表中。使用 Hashtable 时, 指定一个对象作为关键字, 同时指定与该关键字相关联的值。接着该关键字被散列, 而把得到的散列值作为存储在表中的值的下标。

散列表仅仅可以存储重载由 Object 定义的 hashCode() 和 equals() 方法的对象。hashCode() 方法计算和返回对象的散列码。当然, equals() 方法比较两个对象。幸运的是, 许多 Java 内置的类已经实现了 hashCode() 方法。例如, 大多数常见的 Hashtable 类型使用字符串 (String) 对象作为关键字, String 实现 hashCode() 和 equals() 方法。

Hashtable 的构造方法如下。

```

Hashtable()
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map m)

```

第 1 种形式是默认的构造方法。第 2 种形式创建一个散列表, 该散列表具有由 size 指定的原始大小。第 3 种形式创建一个散列表, 该散列表具有由 size 指定的原始大小和由 fillRatio 指定的填充比。填充比必须介于 0.0 和 1.0 之间, 它决定了在散列表向上调整大小之前的充满度。具体地说, 当元素的个数大于散列表的容量乘以它的填充比时, 散列表被扩展。如果没有指定填充比, 则默认使用 0.75。最后, 第 4 种形式创建一个散列表, 该散列表用 m 中的元素初始化。散列表的容量被设为 m 中元素的个数的两倍, 默认的填充因子设为 0.75。第 4 种构造方法是在 Java

2 中新增加的。

除了 Hashtable 目前实现的，由 Map 接口定义的方法之外，Hashtable 定义的从以前版本遗留下来的方法列在下表中。

方法	描述
void clear()	复位并清空散列表
Object clone()	返回调用对象的复制
boolean contains(Object value)	如果一些值与存在于散列表中的 value 相等的话，则返回 true；如果这个值不存在，则返回 false
boolean containsKey(Object key)	如果一些关键字与存在于散列表中的 key 相等的话，则返回 true；如果这个关键字不存在，则返回 false
boolean containsValue(Object value)	如果一些值与散列表中存在的 value 相等的话，则返回 true；如果这个值没有找到，则返回 false（是一种为了保持一致性而在 Java 2 中新增加的非 Map 方法）
Enumeration elements()	返回包含在散列表中的值的枚举
Object get(Object key)	返回包含与 key 相关联的值的对象。如果 key 不在散列表中，则返回一个空对象
boolean isEmpty()	如果散列表是空的，则返回 true；如果散列表中至少包含一个关键字，则返回 false
Enumeration keys()	返回包含在散列表中的关键字的枚举
Object put(Object key, Object value)	将关键字和值插入散列表中。如果 key 已经不在散列表中，则返回 null；如果 key 已经存在于散列表中，则返回与 key 相连的前一个值
void rehash()	增大散列表的大小，并且对其关键字进行再散列
Object remove(Object key)	删除 key 及其对应的值，返回与 key 相关联的值。如果 key 不在散列表中，则返回一个空对象
int size()	返回散列表中的项数
String toString()	返回散列表的等价字符串形式

【范例 16-13】 Hashtable 类的使用（代码 16-13. java）。

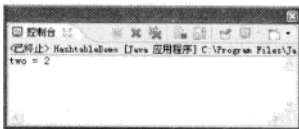
```

01  import java.util.* ;
02  public class HashtableDemo
03  {
04      public static void main(String[] args)
05      {
06          Hashtable numbers = new Hashtable();
07          numbers.put("one", new Integer(1));
08          numbers.put("two", new Integer(2));
09          numbers.put("three", new Integer(3));
10          Integer n = (Integer) numbers.get("two");
11          if (n != null)
12          {
13              System.out.println("two = " + n);
14          }
15      }

```

【运行结果】

保存并运行程序，结果如图所示。



16.5.6 Properties 类

属性 (Properties) 是 Hashtable 的一个子类。它用来保持值的列表，其中关键字和值都是字符串 (String)。Properties 类被许多其他的 Java 类所使用。例如，当获得系统环境值时，System.getProperties() 返回对象的类型。

Properties 定义了下面的实例变量。

Properties defaults;

这个变量包含了一个与属性 (Properties) 对象相关联的默认属性列表。Properties 定义了如下的构造方法。

Properties ()

Properties(Properties propDefault)

第 1 种形式创建一个没有默认值的属性 (Properties) 对象，第 2 种形式创建一个将 propDefault 作为其默认值的对象。在这两种情况下，属性列表都是空的。

除了 Properties 从 Hashtable 中继承下来的方法之外，Properties 自己定义的方法列在下表中。Properties 也包含了一个不被赞成使用的方法：save()。它被 store() 方法所取代，因为它不能正确地处理错误。

方法	描述
String getProperty(String key)	返回与 key 相关联的值。如果 key 既不在列表中，也不在默认属性列表中，则返回一个 null 对象
String getProperty(String key,String defaultProperty)	返回与 key 相关联的值。如果 key 既不在列表中，也不在默认属性列表中，则返回 defaultProperty
void list(PrintStream streamOut)	将属性列表发送给与 streamOut 相链接的输出流
void list(PrintWriter streamOut)	将属性列表发送给与 streamOut 相链接的输出流
void load(InputStream streamIn) throws IOException	从与 streamIn 相链接的输入数据流输入一个属性列表 Enumeration propertyNames() 返回关键字的枚举，也包括那些在默认属性列表中找到的关键字
Object setProperty(String key,String value)	将 value 与 key 关联，返回与 key 关联的前一个值，如果不存在这样的关联，则返回 null (为了保持一致性，在 Java2 中新增加的)
void store(OutputStream streamOut,String description)	在写入由 description 指定的字符串之后，属性列表被写入与 streamOut 相链接的输出流 (在 Java 2 中新增加的)

Properties 类的一个有用的功能是可以指定一个默认属性，如果没有值与特定的关键字相关

联，则返回这个默认属性。例如，默认值可以与关键字一起在 `getProperty()` 方法中被指定——如 `getProperty("name", "default value")`。如果“name”值没有找到，则返回“defaultvalue”。当构造一个 `Properties` 对象时，可以传递 `Properties` 的另一个实例作为新实例的默认值。在这种情况下，如果对一个给定的 `Properties` 对象调用 `getProperty("foo")`，而“foo”并不存在时，Java 在默认 `Properties` 对象中寻找“foo”。它允许默认属性的任意层嵌套。

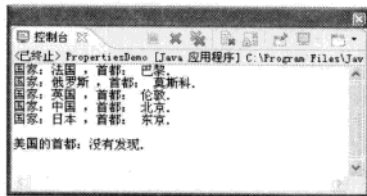
下面的例子说明了 `Properties` 的使用。该程序创建一个属性列表，在其中关键字是各国的名称，值是这些国家的首都。注意试图寻找包括默认值的美国首都时的情况。

【范例 16-14】 `Properties` 的使用（代码 16-14.java）。

```
01  import java.util.*;
02  public class PropertiesDemo
03  {
04      public static void main(String args[])
05      {
06          Properties capitals = new Properties();
07          Set states;
08          String str;
09          capitals.put("中国", "北京");
10          capitals.put("俄罗斯", "莫斯科");
11          capitals.put("日本", "东京");
12          capitals.put("法国", "巴黎");
13          capitals.put("英国", "伦敦");
14
15          // 返回包含映射中项的集合
16          states = capitals.keySet();
17          Iterator itr = states.iterator();
18          while (itr.hasNext())
19          {
20              str = (String) itr.next();
21              System.out.println("国家: " + str + " , 首都: " + capitals.getProperty(str) + ".");
22          }
23          System.out.println();
24          // 查找列表，如果没有则显示为“没有发现”
25          str = capitals.getProperty("美国", "没有发现");
26          System.out.println("美国的首都: " + str + ".");
27      }
28  }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

由于美国不在列表中，所以使用了默认值。尽管当调用 `getProperty()` 方法时，使用默认值是十分有效的，正如上面的程序所展示的那样，但对大多数属性列表的应用来说，有更好的方法去处理默认值。为了展现更大的灵活性，当构造一个属性（`Properties`）对象时，可指定一个默认的属性列表。如果在主列表中没有发现期望的关键词，则会搜索默认列表。

16.5.7 Properties 类中使用 `store()` 和 `load()` 方法

`Properties` 的一个最有用的方面是可以利用 `store()` 和 `load()` 方法方便地对包含在属性（`Properties`）对象中的信息进行存储或从盘中装入信息。在任何时候，都可以将一个属性（`Properties`）对象写入流或从中将其读出。这使得属性列表特别便于实现简单的数据库。

【范例 16-15】 在 `Properties` 类中使用 `store()` 和 `load()` 方法（代码 16-15.java）。

```

01 import java.io.*;
02 import java.util.*;
03 public class PropertiesFile
04 {
05     public static void main(String[] args)
06     {
07         Properties settings = new Properties();
08         try {
09             settings.load(new FileInputStream("c:\\count.java"));
10         } catch (Exception e) {
11             settings.setProperty("count", new Integer(0).toString());
12         }
13         int c = Integer.parseInt(settings.getProperty("count")) + 1;
14         System.out.println("这是本程序第" + c + "次被使用");
15         settings.put("count", new Integer(c).toString());
16         try {
17             settings.store(new FileOutputStream("c:\\count.java"),
18 "PropertiesFile use it .");
19         } catch (Exception e) {
20             System.out.println(e.getMessage());
21         }
22     }
23 }

```



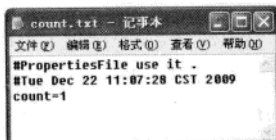
```

21     }
22 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

程序每次启动时都去读取那个记录文件，直接取出文件中所记录的运行次数并加 1 后，又重新将新的运行次数存回文件。由于第 1 次运行时硬盘上还没有那个记录文件，程序去读取那个记录文件时会报出一个异常，就在处理异常的语句中将属性的值设置为 0，表示程序以前还没有被运行过。如果要用到 Properties 类的 store() 方法进行存储，每个属性的关键字和值都必须是字符串类型的，所以上面的程序没有用从父类 Hashtable 继承到的 put、get 方法进行属性的设置与读取，而是直接用了 Properties 类的 setProperty、getProperty 方法进行属性的设置与读取。

类集框架为程序员提供了一个功能强大的设计方案，以完成编程过程中面临的大多数任务。下一次当开发者需要存储和检索信息时，可以考虑使用类集。记住，类集不仅仅是专为那些“大型作业”，例如联合数据库、邮件列表或产品清单系统等所专用的，它们对于一些小型作业也是很有用的。例如，TreeMap 可以给出一个很好的类集，以保留一组文件的字典结构。TreeSet 在存储工程管理信息时是十分有用的。坦白地说，对于采用基于类集的解决方案而受益的问题种类，只受限于开发者的想象力。

16.6 练一练

一、填空题

1. 当将一个不兼容的对象加入一个类集中时，将产生_____异常。
2. _____接口是构造类集框架的基础。
3. 调用_____方法，可以获得集合中的第 1 个对象。
4. 需要获得集合尾部的一个子集合，可以使用_____方法。

二、简答题

简述 ArrayList 的构造方法。

16.7 跟我上机

编写一段程序，使用 TreeSet 类存储以下元素：3、6、9、7、18、25，并输出其元素。

第 17 章

JDK 1.5以上版本的新功能——枚举



本章视频教学录像：49 分钟

枚举是一个被命名的整型常数的集合。枚举在日常生活中常见，例如表示星期的SUNDAY、MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY就是一个枚举。也就是说，事先考虑到某一变量可能取的值，尽量用自然语言中含义清楚的单词来表示它的每一个值，这种方法称为枚举方法，用这种方法定义的类型称为枚举类型。本章将介绍在Java中使用枚举的相关知识。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握枚举的概念
- ☐ 熟悉枚举的作用
- ☐ 掌握 enum 关键字
- ☐ 了解类集对枚举的支持
- ☐ 熟悉枚举中的构造方法
- ☐ 熟悉枚举的接口
- ☐ 掌握枚举的定义方法



17.1 枚举简介

本节视频教学录像：13 分钟

在生活中有很多关于枚举的例子，例如表示星期的 SUNDAY、MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY 就是一个枚举。或者关于月份的，January、February、March、April、MAY、JUNE、JULY 等也是一个枚举。

17.2 枚举的作用

本节视频教学录像：9 分钟

很早在 C 语言中，就存在这样的一个枚举类型，通过此类型可以限制一个内容的取值范围。在 JDK 1.5 之前，Java 语言中并不存在这样枚举的类型，很多之前已经习惯了使用枚举操作的开发人员就感觉很不适应。为了解决这样的问题，在 JDK 1.5 之后，加入了枚举的类型。本节采用一个关于颜色的类来描述枚举的作用与功能，通过对枚举的实际操作了解枚举的作用。

【范例 17-1】 在未出现枚举关键字前如何使用枚举功能。这是在 JDK1.5 以前，枚举没有被运用到 Java 中所采用的一种操作形式。通过此范例可以了解在枚举没有被引用之前，引入类似对象时的情况（代码 17-1. java）。

```
01 public class Color {
02     private String name;
03     public static final Color RED = new Color("红色");
04     public static final Color GREEN = new Color("绿色");
05     public static final Color BLUE = new Color("蓝色");
06     public String getName() {
07         return name;
08     }
09     public void setName(String name) {
10         this.name = name;
11     }
12     private Color(String name) {
13         this.setName(name);
14     }
15     public static Color getInstance(int i) {
16         if (i == 0) {
17             return RED;
18         }
19         if (i == 1) {
20             return GREEN;
21         }
22     }
23 }
```

```

22     if (i == 2) {
23         return BLUE;
24     }
25     return null;
26 }
27 }

```

【代码详解】

第 6~14 行是对 Color 类中构造方法的声明，在新对象的声明中对其的取值进行设置。如在第 3~5 行中，通过对 3 个 Color 的对象声明，可以得到 3 个不同的 Color 对象，分别是 RED、GREEN 和 BLUE，其取值分别为“红色”、“绿色”和“蓝色”。

第 15~27 行设置静态方法 getInstance()。通过此方法，可以通过对不同的数值的调用得到相应的颜色。

【范例分析】

在范例是对构造方法进行私有化的一种操作，在以后的操作中，只能通过 static 属性得到 Color 的实例化对象，或者通过 getInstance()方法取得。在 JDK1.5 以前枚举没有被引入的时候，如果要引入一个 COLOR 类的话，就需要繁琐地将所有的元素枚举出来。通过以上的方式实现的枚举会存在一系列的问题，并且存在操作结果不正确的问题。同时还会造成歧义，影响语句的控制性和可读性。

调用先前所创建的枚举类型，需要通过构造方法 getInstance()取得。由于在 JDK1.5 之前，Java 虚拟机中并没有关于枚举类型的包，所以使用枚举的时候，只能够从自己创造的构造方法中引入。

【范例 17-2】 通过构造方法调用枚举类型。当枚举类型被创建的同时，需要在类型中创建构造方法，以便在今后的使用中调用。在 JDK1.5 之前，枚举类型的创建和调用是非常复杂的（代码 17-2. java）。

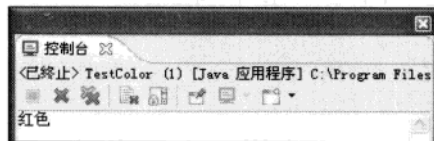
```

01 package org.lxh.enumdemo.demo01;
02 public class TestColor {
03     public static void main(String[] args) {
04         Color c = Color.getInstance(0);
05         System.out.println(c.getName());
06     }
07 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

通过 `getInstance()` 方法调用创建的枚举类型，使得整个程序变得繁琐，并且有碍于程序的可读性和完整性。

早先枚举的使用方法很多，不仅可以通过方法调用，同样也可以在枚举中使用接口。

【范例 17-3】 在枚举中使用接口（代码 17-3. java）。

```
01 package org.lxh.enumdemo.demo01;
02 public interface Color {
03     public static final int RED = 0;
04     public static final int GREEN = 1;
05     public static final int BLUE = 2;
06 }
```

17.3 枚举的用法

枚举在 Java 语言创作中的作用很多，它可以对类似的相同类型的对象同时进行声明，在程序中调用这些元素时，不会出现混淆或重复。由于枚举的统一性，使得开发人员编写的代码量大减少了，同时增加了程序的阅读性和修改性。

枚举是 JDK1.5 之后，Java 语言中新出现的类型，这些类型的出现预示着 Java 的逐渐完善，机器语言的简单化和人性化。相信在不久的将来，机器语言将会同人类语言一样被人们所掌握，成为人与机器交流不可或缺的一部分。

17.3.1 常见的枚举定义方法

在 JDK1.5 之前，人们发现当使用 Java 编写程序的时候，如果需要键入一系列类似的对象，代码量就会增多，并且当对其中的一种对象进行修改或增加时，需要对后来的代码做出同样的修改，以保证程序的严谨性。于是在 JDK1.5 中，Java 也引入了关于枚举类型的设置方法，也是本小节将要了解枚举类型中的重要关键字：Enum。这是本章学习的重点。

在枚举类型中，一般的定义形式如下。

```
enum 枚举名{ 枚举值表 };
```

在枚举值表中应罗列出所有的可用值，这些值也称为枚举元素。

例如，该枚举名为 `weekday`，枚举值共有 7 个，即一周中的 7 天。凡被说明为 `weekday` 类型变量的取值，只能是这 7 天中的某一天。

如同结构和联合一样，枚举变量也可用不同的方式说明，即先定义后说明，同时定义说明或直接说明。

设有变量 `a`、`b`、`c` 被说明为上述的 `weekday`，可采用下述任意一种方式。


```
enum weekday{ sun,mon,tue,wed,thu,fri,sat };
enum weekday a,b,c;
或者为：
enum weekday{ sun,mon,tue,wed,thu,fri,sat }a,b,c;
或者为：
enum { sun,mon,tue,wed,thu,fri,sat }a,b,c
```

上一节中的【范例 17-1】，是在 JDK1.5 以前，通过对静态属性声明的方法创造了一个枚举类型。这样写出来的枚举非常不具有可操作性，也就是说，当修改任何一个对象的时候，需要对多处进行改动。

如果使用 Enum 关键字，代码量将会大大减少。同时，代码的可修改性与可读性也会相应地提高。

接下来通过【范例 17-4】介绍在 JDK1.5 之后，创建一个相同的 Color 类的多种近似对象，需要进行的操作。

【范例 17-4】 枚举在 Java 中的确切含义。本例是在 JDK1.5 之后，也就是当枚举被引入到 Java 中之后，进行创建枚举类型结构的代码。通过本范例与【范例 17-1】对比，可以了解在 Java 改进的过程中，对相同类型对象的不同设置方法（代码 17-4. java）。

```
01 package org.lxx.enumdemo.demo02;
02 public enum Color {
03     RED, GREEN, BLUE;
04 }
```

【代码详解】

除了第 1 行加的 org.lxx.enumdemo.demo02 包声明以外，其余的代码都是对相同 Color 类的不同对象的设置。

在第 2~4 行中，只需要在 Color 类名之前加入关键字 Enum，Java 就可以知道读者创建的是一个关于 Color 类的枚举，其中设置的 RED、GREEN、BLUE 分别是关于 Color 类的对象。通过以上代码，可以看出在 JDK1.5 之后，对于枚举类型的创建，不需要通过【范例 17-1】那样的繁琐过程，就可以完成创建一个枚举类型。

17.3.2 在程序中使用枚举

当创建了一个枚举类型之后，就意味着在今后的代码中进行调用。调用先前定义的枚举类型，同其他的调用语句一样，需要声明该类的一个对象，并通过对象对枚举类型进行操作。

【范例 17-5】 调用全新的枚举类型。本例是关于对【范例 17-4】中创建的 Color 枚举类型的调用程序，程序中会讲解对于通过 Enum 关键字创建的枚举类型的调用方法。运行以下代码将得到一个创建的枚举类型的其中一个取值：RED（代码 17-5. java）。

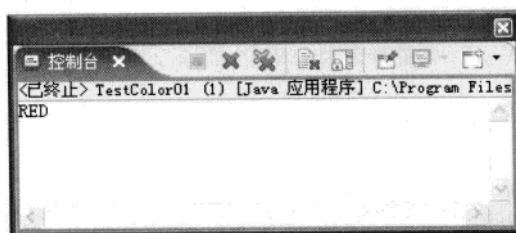
```
01 package org.lxx.enumdemo.demo03;
```



```
02 public class TestColor01 {
03     public static void main(String[] args) {
04         Color c = Color.RED ;           //得到红色
05         System.out.println(c);
06     }
07 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 3~7 行是对于先前声明的 Color 枚举类型的调用。

第 4 行声明了一个新的 Color 类对象 c，将在 Color 类中声明过的 RED 对象赋值给 c。

第 5 行将结果输出在命令台中。

程序运行结果为对象 c 的取值：RED。

通过本范例可以很轻松地调用其中的每一种颜色。而且使用此方式还可以避免之前使用接口实现枚举的操作。

17.3.3 在 Switch 语句中使用枚举

使用 Enum 关键字创建的枚举类型，也可以直接在多处控制语句中使用，如 Switch 语句等。在 JDK1.5 之前，Switch 语句只能用于判断字符或数字，它并不能对在枚举中罗列的内容进行判断和选择。而在 JDK 1.5 之后，通过 Enum 创建的枚举类型，也可以被 Switch 判断使用。

【范例 17-6】 在 Switch 中使用枚举。这是创建的一个 Switch 语句，通过 Switch 调用枚举类型 Color 完成对于枚举的类型的筛选（代码 17-6. java）。

```
01 package org.lxh.enumdemo.demo03;
02 public class TestColor03 {
03     public static void main(String[] args) {
04         switch (Color.RED) {
05             case RED: {
06                 System.out.println("红色");
07                 break;
08             }
09         }
```

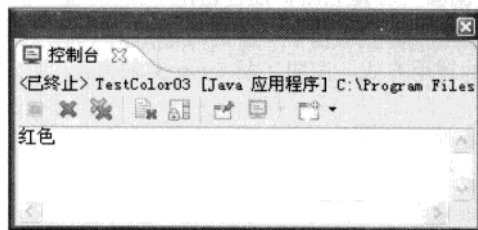
```

09     case GREEN: {
10         System.out.println("绿色");
11         break;
12     }
13     case BLUE: {
14         System.out.println("蓝色");
15         break;
16     }
17 }
18 }
19 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 4~16 行均是对于 Switch 语句的使用，从而说明了在 JDK1.5 之后，Switch 同样可以用来判断一个枚举类型，并对枚举类型做出有效选择。这样在今后的程序写作过程中，就能够大量避免枚举类型多而繁琐的选择问题。这是在 JDK 1.5 之后 Java 新的一种改进，有助于增加代码的可读性和延伸性。

17.4 枚举类和枚举关键字

本节视频教学录像：10 分钟

枚举类型的出现，有助于简洁程序的代码量，减少出错量。在大多数情况下，枚举类和枚举关键字是相互依存的。枚举关键字是定义枚举类型时必不可少的声明，而枚举类则是规定的枚举类型母类。

17.4.1 枚举类

枚举类 (Enum 类) 是在 Java.lang 包下定义的一个公共类，它的作用是用来构造新的枚举类型。这是在 JDK1.5 之后 Java 推出的一个新的类，用来完善关于枚举这一常用集合在 java 中的不足。同时，Enum 类中的构造方法可以大大方便对于代码的操作。在 JDK API 中可以看到，在 Enum 类中定义了大约十多个方法，每一种方法都是对用 Enum 创建的枚举类型的操作，所以

Enum 类指的是一个完整的类型。它拥有自己的方法，当创建了一个关于 Enum 的类型对象的时候，便可以调用其中的方法来完善对于枚举类型的操作。

方法摘要	
protected final clone()	抛出 CloneNotSupportedException。
int compareTo(E e)	比较此枚举与指定对象的顺序。
boolean equals(Object other)	当指定对象等于此枚举常量时，返回 true。
protected void finalize()	枚举类不能有 finalize 方法。
Class<T> getDeclaringClass()	返回与此枚举常量的枚举类型相对应的 Class 对象。
int hashCode()	返回枚举常量的哈希码。
String name()	返回此枚举常量的名称。在其枚举声明中对其进行声明。
int ordinal()	返回枚举常量的序数（它在枚举声明中的位置，其中初始常量序数为零）。
String toString()	返回枚举常量的名称，它包含在声明中。
static <T extends Enum> T valueOf(Class<T> enumType, String name)	返回带指定名称的指定枚举类型的枚举常量。

【范例 17-7】 通过枚举类构造方法得到对象取值范围。本例是通过调用 Enum 类中的方法 values() 来得到枚举类型中各个对象的取值范围。通过实例可以看到在 Enum 类中所包含方法的调用方式，同时了解 Enum 类的作用（代码 17-7. java）。

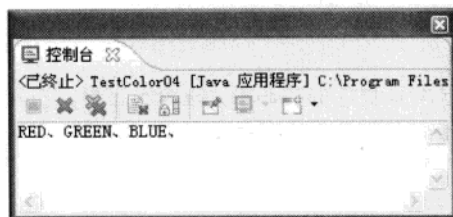
```

01 package org.lxh.enumdemo.demo03;
02 public class TestColor04 {
03     public static void main(String[] args) {
04         for (Color c : Color.values()) {
05             System.out.print(c + "、");
06         }
07     }
08 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

在第 4 行中，For 循环语句中使用的一个 values() 方法，是包含在 Enum 类里面的一种可以得到当前对象取值的方法，通过 For 循环将包含在 Color 枚举类型中的对象依次输出。

实际上，Enum 类里还包含有很多种方法，对于不同方法的作用，需要读者在程序中深入了解。

17.4.2 枚举关键字

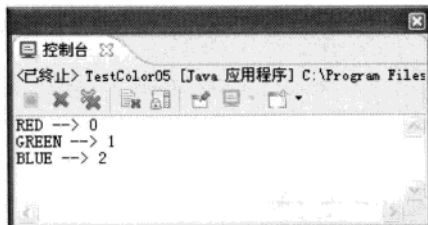
相对于枚举类 (Enum 类), 枚举关键字 (enum 关键字) 则是定义的一个枚举类型。实际上, 在此次定义的过程中, 通过 enum 关键字相当于定义了一个类, 并且此类将继承 Enum 类。被定义的对象拥有 Enum 类中的构造方法的使用权。在 Enum 类中的构造方法是受保护的, 实际上对于每一个枚举的对象一旦声明之后, 就表示自动调用此构造方法, 所有的编号方式均采用自动编号的方式进行。

【范例 17-8】 通过调用 Enum 类中的 ordinal() 方法, 输出枚举类型中每一个对象的编号。在没有对编号做出特殊声明时, Java 虚拟机一般将对被创建的枚举类型对象自动编号, 编号从 0 开始 (代码 17-8. java)。

```
01 package org.lxh.enumdemo.demo03;
02 public class TestColor05 {
03     public static void main(String[] args) {
04         for (Color c : Color.values()) {
05             System.out.println(c.name() + " --> " + c.ordinal());
06         }
07     }
08 }
```

【运行结果】

保存并运行程序, 结果如图所示。



【代码详解】

第 4 行通过 For 循环完成对 Color 类中的对象的逐个调用。

第 5 行引入了在 Enum 类中包含的两种构造方法: name() 和 ordinal(), 然后将不同的对象名称按照其编号输出。

通过本范例可以很清楚地得到关于在【范例 17-1】中定义的 Color 枚举类型中各项元素的编号。

17.4.3 枚举类与枚举关键字的区别

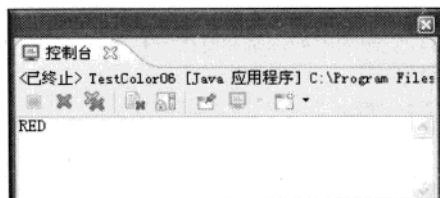
在 JDK API 中, 找到 Java.lang 下的 Enum 类, 可以看到许多关于 Enum 类的构造方法。但这些方法在 Java 中是受保护的, 也就是说, 可以直接调用其中的方法。

【范例 17-9】 调用 Enum 类中的 valueOf() 方法，了解如何对 Enum 类中受保护方法的调用和控制（代码 17-9. java）。

```
01 package org.lxx.enumdemo.demo03;
02 public class TestColor06 {
03     public static void main(String[] args) {
04         Color c = Color.valueOf(Color.class, "RED");
05         System.out.println(c);
06     }
07 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

在第 4 行中，调用在 Enum 类中的方法 valueOf()，可以得到当前枚举类型的取值。但定义这个方法的构造是非常繁琐的。

```
public static <T extends Enum<T>> T valueOf(Class<T> enumType,String name)
```

可以通过一个 Color.class 类直接传入，从而得到结果。通过以上的代码可以看到，直接通过枚举取得指定内容则会更加简单。

17.5 类集对于枚举的支持

本节视频教学录像：8 分钟

在 JDK1.5 之后，Java 中又新增加了两个类集的操作类，分别是 EnumMap 和 EnumSet。

17.5.1 EnumMap

Map 是 Java 中的一个类，如同其他的类一样，EnumMap 是 Map 接口的子类，操作的形式与 Map 是一致的。可以通过调用 EnumMap 的方法来实现对于对象的控制。

【范例 17-10】 使用 EnumMap 操作类。定义一个关于 EnumMap 方法的程序，了解怎样使用 EnumMap 这样一个类集的操作类（代码 17-10. java）。

```
01 package org.lxx.enumdemo.demo04;
```



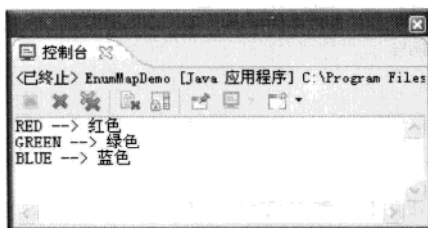
```

02 import java.util.EnumMap;
03 import java.util.Map;
04 public class EnumMapDemo {
05     public static void main(String[] args) {
06         EnumMap<Color, String> eMap = new EnumMap<Color, String>(Color.class);
07         eMap.put(Color.RED, "红色");
08         eMap.put(Color.GREEN, "绿色");
09         eMap.put(Color.BLUE, "蓝色");
10         for (Map.Entry<Color, String> me : eMap.entrySet()) {
11             System.out.println(me.getKey() + " --> " + me.getValue());
12         }
13     }
14 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 2~3 行，引入了在 Java 中定义的包 lang 的 EnumMap 类和 Map 类。

第 6 行创建了一个 EnumMap 类的对象 eMap。

第 7~9 行，分别将语句“红色”、“绿色”、“蓝色”赋值给对应的枚举类型元素 RED、GREEN、BLUE。Put()方法是定义在 EnumMap 类中的构造方法。

第 10~14 行，通过 For 循环将所拥有元素的名称和取值输出。其中，EntrySet()是定义在 Map 类中的一个构造方法，作用是返回此映射中包含的映射关系的 Set 视图。

17.5.2 EnumSet

EnumSet 本身是 Set 接口的子类，但是在此类中并没有任何的构造方法定义，表示构造方法被私有化。同时，所有对于此类的方法的操作均是静态的操作。

【范例 17-11】 测试 EnumSet 静态方法。测试在 EnumSet 中定义的静态方法，通过对定义在 EnumSet 中方法的调用，了解 EnumSet 的特性和作用（代码 17-11.java）。

```

01 package org.lxh.enumdemo.demo04;
02 import java.util.EnumSet;

```

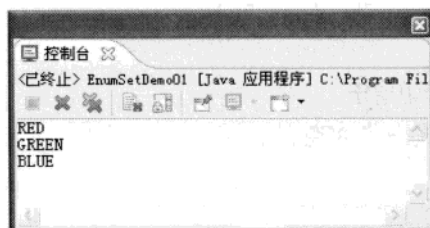
```

03 import java.util.Iterator;
04 public class EnumSetDemo01 {
05     public static void main(String[] args) {
06         EnumSet<Color> eSet = EnumSet.allOf(Color.class);    //表示将全部的内容设置到集合
07         Iterator<Color> iter = eSet.iterator();
08         while(iter.hasNext()){
09             System.out.println(iter.next());
10         }
11     }
12 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

以上代码调用了 `allOf()` 的静态内部方法，用来将全部的内容设置到集合，也就是说将枚举类型所包含的内容或者取值全部调出。

包含在 `EnumSet` 类中的方法，不仅可以将全部的内容设置到集合中，同时也可以设置到一个空集合当中。

【范例 17-12】 调用 `noneOf()` 方法。对同样包含在 `EnumSet` 类中的不同方法的调用，来测试 `EnumSet` 方法（代码 17-12. java）。

```

01 package org.lxh.enumdemo.demo04;
02 import java.util.EnumSet;
03 import java.util.Iterator;
04 public class EnumSetDemo02 {
05     public static void main(String[] args) {
06         EnumSet<Color> eSet = EnumSet.noneOf(Color.class);    // 表示此类型的空集合
07         Iterator<Color> iter = eSet.iterator();
08         while(iter.hasNext()){
09             System.out.println(iter.next());
10         }
11     }
12 }

```

【范例分析】

使用 `noneof()` 方法, 在这里表示此类型的空集合。也就是说, 在这里运行后将得不到任何输出结果。

17.6 深入了解枚举

▶ 本节视频教学录像: 9 分钟

枚举的作用在 Java 中, 甚至在所有的计算机高级语言中, 都占有举足轻重的地位。了解枚举不能够浅尝辄止。将枚举熟练化, 是提高工程水平及工程逻辑度的有效手段。

17.6.1 枚举的构造方法

枚举的使用非常灵活。枚举可以应用于代码中的各个角落, 只要定义的对象具有枚举的形式, 均可以使用枚举对其进行定义, 这样在减少代码量的同时, 也可增加代码的可读性和可操作性。在枚举中也可以直接定义构造方法, 但需要注意的是: 一旦构造方法定义之后, 则所有的枚举对象都必须明确地调用此构造方法。

【范例 17-13】 定义枚举的构造方法。对枚举类型 `Color` 定义了两个构造方法, 分别是 `getName()` 和 `setName()`, 在枚举中定义构造方法需要完全按照之后定义的构造方法进行调用 (代码 17-13. java)。

```
01 package org.lxh.enumdemo.demo05;
02 public enum Color {
03     RED("红色"), GREEN("绿色"), BLUE("蓝色");
04     private String name;
05     public String getName() {
06         return name;
07     }
08     public void setName(String name) {
09         this.name = name;
10     }
11     Color(String name) {
12         this.setName(name);
13     }
14 }
```

【代码详解】

在前期声明的枚举类型中, 要完全按照之后定义的构造方法进行调用。

第 3 行中的 “`RED("红色"), GREEN("绿色"), BLUE("蓝色")`” 为新创建的枚举类型的元素。由于在第 8~9 行中定义了枚举的构造方法, 所以在声明枚举类型的同时, 要按照后来定义的 `setName()` 方法进行声明。

而且在枚举中的构造是不能声明为 `public` 的，因为外部是不能调用枚举的构造方法的。枚举的构造方法都是内部的静态方法。如程序中所示，此时，每一个枚举对象都有其自己的 `name` 属性。

17.6.2 枚举的接口

当一个枚举实现一个接口之后，各个枚举对象都必须分别实现接口中的抽象方法。

【范例 17-14】 创建枚举接口。本例是新创建的一个接口，其中声明了一个方法 `getColor()`（代码 17-14. java）。

```
01 package org.lxx.enumdemo.demo06;
02     public interface Info {
03         public String getColor();
04     }
```

【范例分析】

新设置一个 `INFO` 的接口，用来调用 `getcolor()` 方法，从而得到一个枚举类中的内容。

【范例 17-15】 对新建接口抽象方法的调用。对接口中定义的抽象方法的调用，并且在后期声明的枚举类型中，任何一个对象都必须分别实现接口中的抽象方法（代码 17-15. java）。

```
01 package org.lxx.enumdemo.demo06;
02     public enum Color implements Info { // 实现接口
03         RED {
04             public String getColor() {
05                 return "红色";
06             }
07         },
08         GREEN {
09             public String getColor() {
10                 return "绿色";
11             }
12         },
13         BLUE {
14             public String getColor(){
15                 return "蓝色";
16             }
17         };
18     }
```

【代码详解】

第 3~17 行在调用枚举时，即调用 RED、GREEN、BLUE 等的时候，需要在后面实现在接口中定义的方法 `getColor()`，只有这样，才能在枚举中去实现接口。而不能直接在后面调用 RED(“红色”)这样一种方式。

【范例 17-16】 通过代码测试上面实现的接口类型（代码 17-16.java）。

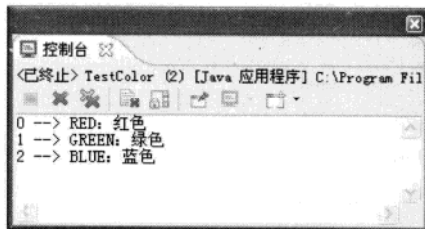
```

01 package org.lxx.enumdemo.demo06;
02 public class TestColor {
03     public static void main(String[] args) {
04         for (Color c : Color.values()) {
05             System.out.println(c.ordinal() + " --> " + c.name() + ": " + c.getColor());
06         }
07     }
08 }

```

【运行结果】

保存并运行程序，结果如图所示。

**【范例分析】**

通过测试可以看到，定义在枚举中的抽象方法必须在后期声明的枚举类型中，任何一个对象都分别实现接口中的抽象方法后才能测试成功。

17.6.3 在枚举中定义抽象方法

通过枚举实现接口，同样，Java 也可以在枚举中直接定义抽象方法。在一个枚举中可以定义一个或多个抽象方法。需要注意的是：枚举中的每个对象都必须单独地实现此方法。

【范例 17-17】 定义枚举的抽象方法。直接在枚举中定义抽象方法，其中每个对象都必须单独实现此方法（代码 17-17.java）。

```

01 package org.lxx.enumdemo.demo06;
02 public enum Color {
03     RED {
04         public String getColor() {
05             return "红色";

```



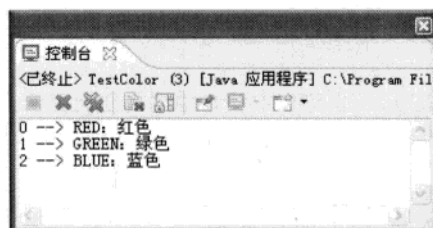
```
06 }
07 },
08 GREEN {
09     public String getColor() {
10         return "绿色";
11     }
12 },
13 BLUE {
14     public String getColor(){
15         return "蓝色";
16     }
17 };
18     public abstract String getColor();
19 }
```

【范例 17-18】 测试在枚举汇总定义的抽象方法。本例是对【范例 17-14】的测试，通过测试可以得到在枚举中直接定义的抽象方法（代码 17-18. java）。

```
01 package org.lxh.enumdemo.demo06;
02 public class TestColor {
03     public static void main(String[] args) {
04         for (Color c : Color.values()) {
05             System.out.println(c.ordinal() + " --> " + c.name() + ": " + c.getColor());
06         }
07     }
08 }
```

【运行结果】

保存并运行程序，结果如图所示。



17.7 练一练

一、填空题

1. 在定义枚举的过程中，使用关键字_____进行定义。

2. 在枚举类中，将全部内容设置到集合的方法是_____。

二、简答题

1. 简述什么是枚举？
2. 简述枚举类与枚举关键字的区别。

17.8 跟我上机

下面的代码，是在枚举中定义的抽象方法 `getDay()`。要求找出代码中的错误。

```
01 public enum Day {  
02     MONDAY {  
03         return "星期一";  
04     },  
05     TUESDAY {  
06         public String getDay() {  
07             return "星期二";  
08         }  
09     },  
10     WEDNESDAY {  
11         return "星期三";  
12     };  
13     public abstract String getDay();  
14 }
```

第 18 章

给编译器看的注释——Annotation



本章视频教学录像：1 小时 6 分钟

Annotation功能建立在反射机制之上，通过Annotation可以对程序进行注释操作。本章讲解系统内建的Annotation、如何自定义Annotation、反射和Annotation的关联，以及如何通过Annotation生成Documented注释。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握 Annotation 的相关概念
- ☐ 熟悉自定义 Annotation 的方法
- ☐ 了解反射和 Annotation 的关联



18.1 Annotation

▶ 本节视频教学录像：4 分钟

Annotation 实际上表示的是一种注释的语法。在 Java 中最早的程序是提倡程序与配置代码相分离，而最新的理论是将所有的配置直接写入到程序之中，如果想要完成这样的功能，就要使用 Annotation。

18.2 系统内建的 Annotation

▶ 本节视频教学录像：14 分钟

在 JDK 1.5 之后的系统中，内建了 3 个 Annotation：@Override、@Deprecated、@SuppressWarnings。

18.2.1 @Override

表示进行正确的覆写操作。例如有如下的两个类。

```
package org.lxx.demo01;

public class Person {
    public String say(){
        return "人在说话。";
    }
}
```

现在要求增加此类的子类，并且在子类中覆写 say() 方法。

```
package org.lxx.demo01;

public class Studnt extends Person {
    public String say(){
        return "学生在说话。";
    }
}
```

此时，代码已经正确地覆写成功了。如果现在的名称编写错误了，则肯定不叫覆写。那么此时在系统中为了保证程序可以正确地进行覆写的操作，在覆写的时候可以明确地使用 @Override 表示方法是属于覆写的操作。

```
package org.lxx.demo01;

public class Studnt extends Person {
    @Override
```

```
public String say(){
    return "学生在说话。";
}
}
```

18.2.2 @Deprecated

@Deprecated 注释表示是不建议使用的操作。例如之前在讲解线程的时候，曾经讲解过线程中的 stop()、resume()、suspend()等方法是不建议使用的。

```
package org.lxx.demo02;
public class Info {
    @Deprecated
    public String getInfo(){
        return "hello";
    }
}
```

使用@Deprecated 声明只是不建议使用，如果使用的话，只会出现警告信息而已。

```
package org.lxx.demo02;
public class TestInfo {
    public static void main(String[] args) {
        new Info().getInfo();
    }
}
```

18.2.3 @SuppressWarnings

@SuppressWarnings 表示的是压制警告。如果有一些警告信息则可压制掉，不出现警告的提示。

```
package org.lxx.demo02;
public class TestInfo {
    @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        new Info().getInfo();
    }
}
```


使用 SuppressWarnings 操作的时候可以同时压制多个警告。

通过 SuppressWarnings 的类，可以发现在此类中存在一个 value 的字符串数组。

```
package org.lxx.demo03;
import java.io.Serializable;
@SuppressWarnings({ "serial", "deprecation" })
public class Person extends Info implements Serializable {
}
```

Person 类中既继承了 Info 类，又实现了序列化的接口，所以此时会出现多个警告。

或者可以明确地表示，是为 SuppressWarnings 中的 value 属性赋值。

```
package org.lxx.demo03;
import java.io.Serializable;
@SuppressWarnings(value = { "serial", "deprecation" })
public class Person extends Info implements Serializable {
}
```

18.3 自定义 Annotation

▶ 本节视频教学录像：10 分钟

定义 Annotation 的语法如下。

```
public @interface Annotation 的名称 {}
```

下面按照此格式定义一个简单的 Annotation。

```
package org.lxx.demo04;
public @interface MyAnnotation {
}
```

如果现在要使用此 Annotation 的话，若不在同一个包中，则需要导入，导入之后使用@的形式反引，语法：@MyAnnotation。

```
package org.lxx.demo04;
@MyAnnotation
public class Info {
}
```

在一个 Annotation 中实际上也可以定义若干个属性。

```
package org.lxx.demo04;
public @interface MyAnnotation {
```

```
public String key();  
public String value();  
}
```

在此 Annotation 中定义了 key 和 value 两个变量，若此时要想使用 Annotation 的话，则必须明确地给出具体值的内容。

```
package org.lxh.demo04;  
@MyAnnotation(key = "MLDN", value = "www.mldnjava.cn")  
public class Info {  
}
```

如果现在希望为一个 Annotation 设置默认内容的话，可以通过 default 完成。

```
package org.lxh.demo06;  
public @interface MyAnnotation {  
    public String key() default "LXH";  
    public String value() default "HAHA";  
}
```

如果没有设置内容，则将默认值取出继续使用。
Annotation 中的变量的内容可以通过枚举指定范围。

```
package org.lxh.demo07;  
public enum Color {  
    RED, GREEN, BLUE;  
}
```

那么以上的枚举将限制 Annotation 的使用变量的取值。

```
package org.lxh.demo07;  
public @interface MyAnnotation {  
    public String key() default "LXH";  
    public String value() default "HAHA";  
    public Color color() default Color.RED;  
}
```

如果现在在 Info 类中使用此 Annotation 的时候并没有指定其规定范围的内容，则会出现错误。

```
package org.lxh.demo07;  
@MyAnnotation(color = "red")  
public class Info {  
}
```

Annotation 中的变量还可以使用一个数组表示。

```
package org.lxx.demo08;

public @interface MyAnnotation {
    public String key() default "LXH";
    public String value() default "HAHA";
    public Color color() default Color.RED;
    public String[] url();
}
```

以后在使用的时候必须按照数组的方式进行操作。

```
package org.lxx.demo08;

@MyAnnotation(url = { "www.mldn.cn", "www.mldnjava.cn" })
public class Info {
}
```

18.4 Retention 和 RetentionPolicy

▶ 本节视频教学录像：4 分钟

在 java.lang.annotation 包中定义了所有的与 Annotation 有关的操作，先观察 Retention。Retention 本身是一个 Annotation，其中的取值是通过 RetentionPolicy 这个枚举类型指定的范围。

在 RetentionPolicy 中规定了以下 3 个作用范围。

- (1) 只在源代码中起作用：public static final RetentionPolicy SOURCE。
- (2) 只在编译之后的 class 中起作用：public static final RetentionPolicy CLASS。
- (3) 在运行的时候起作用：public static final RetentionPolicy RUNTIME。

如果一个 Annotation 要想起作用，则必须使用 RUNTIME 范围。

任何一个自定义的 Annotation 都是继承了 java.lang.annotation.Annotation 接口。

18.5 反射与 Annotation

▶ 本节视频教学录像：18 分钟

一个 Annotation 如果要想起作用，则肯定要依靠反射机制。通过反射可以取得在一个方法上声明的 Annotation 的全部内容。

在 Filed、Method、Constructor 的父类上定义了以下与 Annotation 反射操作相关的方法。

- (1) 取得全部的 Annotation：public Annotation[] getAnnotations()。
- (2) 判断操作的是否是指定的 Annotation。

```
|- public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)
```

18.5.1 取得全部的 Annotation

现在在一个方法上使用 3 个内建的 Annotation 声明。

```
package org.lxh.demo09;

public class Info {
    @Override
    @Deprecated
    @SuppressWarnings(value="")
    public String toString() {
        return "hello" ;
    }
}
```

这 3 个内建的 Annotation 中只有 @Deprecated 是 RUNTIME 类型。实践证明，如果在程序运行的时候当取得了全部的 Annotation 时，只能取得一个。

```
package org.lxh.demo09;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

public class ClassAnnotationDemo01 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.lxh.demo09.Info");
        Method toStringMethod = cls.getMethod("toString");
        Annotation ans[] = toStringMethod.getAnnotations();
        // 取得全部的 Annotation
        for (int i = 0; i < ans.length; i++) {
            System.out.println(ans[i]);
        }
    }
}
```

此代码很好地证明了只有在 Runtime 范围中的 Annotation 才可以被用户找到。

18.5.2 加入自定义的 Annotation

在一个自定义的 Annotation 编写的时候，如果要想让其有意义，则必须使用 RUNTIME 声明范围。

```
package org.lxh.demo09;
import java.lang.annotation.*;

@Retention(value=RetentionPolicy.RUNTIME)
```

```
public @interface MyAnnotation {
    public String key() default "LXH";
    public String value();
}
```

之后在 Info 类中增加此 Annotation。

```
package org.lxx.demo09;
public class Info {
    @Override
    @Deprecated
    @SuppressWarnings(value = "")
    @MyAnnotation(key = "MLDN", value = "www.mldnjava.cn")
    public String toString() {
        return "hello";
    }
}
```

在使用的时候真正需要取得的是向自定义 Annotation 中设置的内容。

```
package org.lxx.demo09;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
public class ClassAnnotationDemo02 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.lxx.demo09.Info");
        Method toStringMethod = cls.getMethod("toString");
        Annotation ans[] = toStringMethod.getAnnotations(); // 取得全部的 Annotation
        for (int i = 0; i < ans.length; i++) {
            if (toStringMethod.isAnnotationPresent(MyAnnotation.class)) {
                MyAnnotation my = null; // 声明 Annotation 的对象
                my = toStringMethod.getAnnotation(MyAnnotation.class);
                String key = my.key();
                String value = my.value();
                System.out.println(key + " --> " + value);
            }
        }
    }
}
```

在实际开发中，不用去关心这些低层的操作原理，在程序使用中都会为其提供支持。

18.6 深入 Annotation

▶ 本节视频教学录像：16 分钟

在 `java.lang.annotation` 中存在以下的 Annotation：Target、Documented、Inherited。

18.6.1 Target

一个自定义的 Annotation 可以在任意的位置上使用。

```
package org.lxx.demo11;

@MyAnnotation
public class Info {
    @MyAnnotation
    private String name ;
    @MyAnnotation
    public String toString() {
        return "hello" ;
    }
}
```

因为可以在任意的位置上使用，因此在操作的时候就会出现一些问题，例如一些 Annotation 只希望在方法的声明上使用。那么此时，就必须设置 Annotation 的作用范围。

在 @Target 注释中，存在 ElementType 类型的变量，在此变量中存在 7 种范围。

- (1) 只能在 Annotation 中出现：public static final ElementType ANNOTATION_TYPE。
- (2) 只能在构造方法中出现：public static final ElementType CONSTRUCTOR。
- (3) 本地变量上使用：public static final ElementType LOCAL_VARIABLE。
- (4) 只能在方法上使用：public static final ElementType METHOD。
- (5) 在参数声明上使用：public static final ElementType PARAMETER。
- (6) 在包的声明上使用：public static final ElementType PACKAGE。
- (7) 只能在类或接口上使用：public static final ElementType TYPE。

```
package org.lxx.demo11;
import java.lang.annotation.*;

@Target(value=ElementType.METHOD)
@Retention(value=RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    public String key() default "LXH";
    public String value() default "LXH";
}
```

此 Annotation 只能在方法上使用。或者可以同时指定多个范围。

```

package org.lxh.demo11;
import java.lang.annotation.*;
@Target(value = { ElementType.METHOD, ElementType.TYPE })
@Retention(value = RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    public String key() default "LXH";
    public String value() default "LXH";
}

```

18.6.2 Documented 注释

此种表示的是文档的注释格式。

```

package org.lxh.demo12;
import java.lang.annotation.*;
@Documented
@Target(value = { ElementType.METHOD, ElementType.TYPE })
@Retention(value = RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    public String key() default "LXH";
    public String value() default "LXH";
}

```

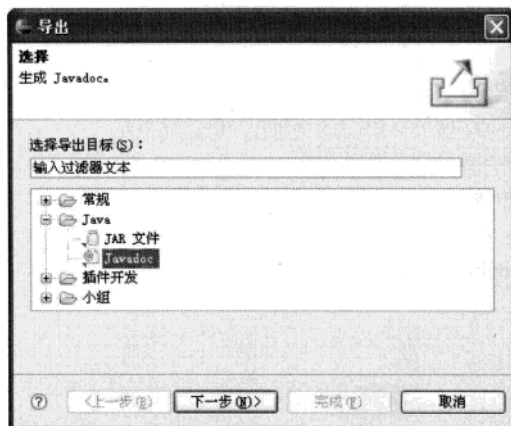
在使用类中可以加入文档注释。

```

package org.lxh.demo12;
@MyAnnotation
public class Info {
    private String name ;
    /**
     * 本方法是覆写 Object 类中的 toString()方法
     */
    @MyAnnotation
    public String toString() {
        return "hello" ;
    }
}

```

加入文档注释之后，程序的最大好处是可以在 doc 文档中出现。
在 Eclipse 中直接支持 javadoc 文档的生成。



18.6.3 Inherited

表示一个 Annotation 能否被使用其类的子类继续继承下去。如果没有写上此注释，则此 Annotation 根本就是无法继承的。

```
package org.lxh.demo13;
import java.lang.annotation.*;

@Inherited
@Documented
@Target(value = { ElementType.METHOD, ElementType.TYPE })
@Retention(value = RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    public String key() default "LXH";
    public String value() default "LXH";
}
```

此 Annotation 可以被子类继承。

18.7 练一练

一、填空题

1. 在 JDK1.5 之后，系统中内建的 3 个 Annotation 分别为_____、_____和_____。
2. 定义一个名称为“firsttip”的 Annotation 的语句为_____。

二、简答题

简述在 RetentionPolicy 中规定的 3 个作用范围。

第 3 篇

高级应用

在本篇中，将结合实例程序学习 Java 开发中的一些高级技术。例如 Java 多线程机制、文件 IO 操作、Java 网络程序设计以及 Java 数据库编程等高级开发技术。

- ▶ 第 19 章 齐头并进完成任务——多线程
- ▶ 第 20 章 文件 IO 操作
- ▶ 第 21 章 Java 网页小程序——Java Applet
- ▶ 第 22 章 Java 网络程序设计
- ▶ 第 23 章 Java 数据库编程

第 19 章

齐头并进完成任务——多线程



本章视频教学录像：2 小时

采用Java中的多线程机制可以使计算机资源得到更充分的使用，多线程可以使程序在同一时间内完成很多操作。本章讲解进程与线程的共同点和区别、实现多线程的方法、线程的状态、对线程操作的方法、多线程的同步、线程间的通信，以及线程生命周期的控制等内容。

本章要点（已掌握的在方框中打勾）

- ☐ 了解进程与线程
- ☐ 掌握实现多线程的方法
- ☐ 熟悉线程的状态
- ☐ 熟悉线程操作的方法
- ☐ 熟悉线程同步的方法
- ☐ 熟悉线程间通信的方法



Java 是少数的几种支持“多线程”的语言之一。大多数的程序语言只能循序运行单独的一个程序块,但无法同时运行不同的多个程序块。Java 的“多线程”恰可弥补这个缺憾,它可以让不同的程序块一起运行,如此一来就可让程序运行得更为顺畅,同时也可达到多任务处理的目的。

19.1 进程与线程

▶ 本节视频教学录像: 6 分钟

进程的特征是:

- (1) 一个进程就是一个执行中的程序,而每一个进程都有自己独立的一块内存空间,一组系统资源。在进程概念中,每一个进程的内部数据和状态都是完全独立的。
- (2) 创建并执行一个进程的系统开销是比较大的。
- (3) 进程是程序的一次执行过程,是系统运行程序的基本单位。

线程的特征是:

- (1) 在 Java 中,程序通过流控制来执行程序流。程序中单个顺序的流控制称为线程。
- (2) 多线程指的是在单个进程中可以同时运行多个不同的线程,执行不同的任务。多线程意味着一个程序的多行语句可以看上去几乎同时运行。

二者都是一段完成某个特定功能的代码,是程序中单个顺序的流控制。

不同的是同类的多个线程是共享一块内存空间和一组系统资源,而线程本身的数据通常只有微处理器的寄存器数据,以及一个供程序执行时使用的堆栈。所以系统在产生一个线程,或者在各个线程之间切换时,负担要比进程小得多,正因如此,线程也被称为轻负荷进程(light-weight process)。一个进程中可以包含多个线程。

多线程是实现并发机制的一种有效手段。进程和线程一样,都是实现并发的一个基本单位。线程和进程的主要差别体现在以下两个方面。

- (1) 同样作为基本的执行单元,线程是划分得比进程更小的执行单位。
- (2) 每个进程都有一段专用的内存区域。与此相反,线程却共享内存单元(包括代码和数据),通过共享的内存单元来实现数据交换、实时通信与必要的同步操作。

多线程的应用范围很广。在一般情况下,程序的某些部分同特定的事件或资源联系在一起,同时又不想为它而暂停程序其他部分的执行,在这种情况下,就可以考虑创建一个线程,令它与那个事件或资源关联到一起,并让它独立于主程序运行。通过使用线程,可以避免用户在运行程序和得到结果之间的停顿,还可以让一些任务(如打印任务)在后台运行,而用户则在前台继续完成一些其他的工作。总之,利用多线程技术,可以使编程人员方便地开发出能同时处理多个任务的功能强大的应用程序。

19.2 认识线程

▶ 本节视频教学录像: 37 分钟

在传统的程序语言里,运行的顺序总是必须顺着程序的流程来走,遇到 if...else 语句就加以判断,遇到 for、while 等循环就会多绕几个圈,最后程序还是按着一定的程序走,且一次只能运行一个程序块。

Java 的“多线程”打破了这种传统的束缚。所谓的线程(Thread)是指程序的运行流程,“多

线程”的机制则是指可以同时运行多个程序块，使程序运行的效率变得更高，也可克服传统程序语言所无法解决的问题。例如，有些包含循环的线程可能要使用比较长的一段时间来运行，此时便可让另一个线程来做其他的处理。

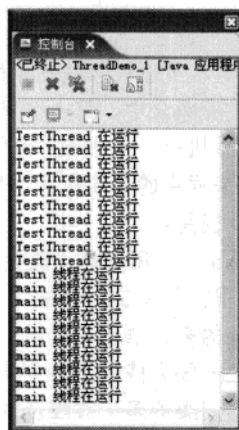
本节将用一个简单的程序来说明单一线程与多线程的不同。ThreadDemo_1 是单一线程的范例，其程序代码的编写方法与前面的程序代码并没有什么不同。

【范例 19-1】 线程使用范例（代码 19-1.txt）。

```

01 public class ThreadDemo_1
02 {
03     public static void main(String args[])
04     {
05         new TestThread().run();
06         // 循环输出
07         for(int i=0;i<10;i++)
08         {
09             System.out.println("main 线
程在运行");
10         }
11     }
12 }
13 class TestThread
14 {
15     public void run()
16     {
17         for(int i=0;i<10;i++)
18         {
19             System.out.println("TestThread 在运行");
20         }
21     }
22 }

```



【代码详解】

第 15~21 行定义了 run() 方法，用于循环输出 10 个连续的字符串。

第 5 行创建 TestThread 对象之后调用 run() 方法，输出“TestThread 在运行”，最后执行 main 方法中的循环，输出“main 线程在运行”。

【范例分析】

从本例中可看出，要想运行 main 方法中的循环，必须要等 TestThread 类中的 run() 方法执行完后才可以运行，这便是单一线程的缺陷。在 Java 里，是否可以同时运行第 9 行与 19 行的语句，使得“main 线程在运行”和“TestThread 在运行”交替输出呢？答案是肯定的，其方法是——在 Java 里激活多个线程。

【运行结果】

保存并运行程序，结果如图所示。

如何激活线程？

如果在类里要激活线程，必须先做好下面两个准备。

- (1) 线程必须扩展自 Thread 类，使自己成为它的子类。
- (2) 线程的处理必须编写在 run() 方法内。

19.2.1 通过继承 Thread 类实现多线程

Thread 存放在 java.lang 类库里，但并不需要加载 java.lang 类库，因为它会自动加载。此外，run()方法是定义在 Thread 类里的一个方法，因此把线程的程序代码编写在 run()方法内，事实上所做的就是覆写的操作。因此要使一个类可激活线程，必须按照下面的语法来编写。

```
class 类名称 extends Thread           // 从 Thread 类扩展出子类
{
    属性...
    方法...
    修饰符 run(){                      // 覆写 Thread 类里的 run()方法
        以线程处理的程序;
    }
}
```

接下来按照上述的语法来重新编写 ThreadDemo_1，使它可以同时激活多个线程。

【范例 19-2】 同时激活多个线程（代码 19-2.txt）。

```
01  public class ThreadDemo_1
02  {
03      public static void main(String args[])
04      {
05          new TestThread().start();
06          // 循环输出
07          for(int i=0;i<10;i++)
08          {
09              System.out.println("main 线程在运行");
10          }
11      }
12  }
13  class TestThread extends Thread
14  {
15      public void run()
16      {
17          for(int i=0;i<10;i++)
18          {
19              System.out.println("TestThread 在运行");
20          }
21      }
22  }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

从运行结果中可以看到两行输出是交替进行的，也就是说程序是采用多线程机制运行的。与之前的程序相比，修改后的程序的第 13 行 TestThread 类继承了 Thread 类，第 5 行调用的不再是 run() 方法，而是 start() 方法。所以，要启动线程必须调用 Thread 类之中的 start() 方法，而调用了 start() 方法，也就是调用了 run() 方法。

19.2.2 通过实现 Runnable 接口实现多线程

从前面的章节中读者应该已经清楚，JAVA 程序只允许单一继承，即一个子类只能有一个父类，所以在 Java 中如果一个类继承了某一个类，同时又想采用多线程技术的时候，就不能用 Thread 类产生线程，因为 Java 不允许多继承，这时要用 Runnable 接口来创建线程。多线程的定义语法如下。

```
class 类名称 implements Runnable           // 实现 Runnable 接口
{
    属性...
    方法...
    修饰符 run(){                           // 覆写 Thread 类里的 run() 方法
        以线程处理的程序;
    }
}
```

【范例 19-3】 用 Runnable 接口实现多线程使用实例 1（代码 19-3.txt）。

```
01 public class ThreadDemo_2
02 {
```

322

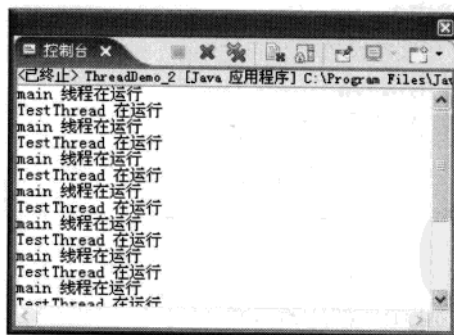

```

03     public static void main(String args[])
04     {
05         TestThread t = new TestThread();
06         new Thread(t).start();
07         // 循环输出
08         for(int i=0;i<10;i++)
09         {
10             System.out.println("main 线程在运行");
11         }
12     }
13 }
14 class TestThread implements Runnable
15 {
16     public void run()
17     {
18         for(int i=0;i<10;i++)
19         {
20             System.out.println("TestThread 在运行");
21         }
22     }
23 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 14 行中的 TestThread 类实现了 Runnable 接口，同时覆写了 Runnable 接口之中的 run() 方法，也就是说此类为一个多线程实现类。

第 5 行实例化了一个 TestThread 类的对象。

第 6 行通过 TestThread 类（Runnable 接口的子类）去实例化一个 Thread 类的对象，之后调用 start() 方法启动多线程。

【范例分析】

从输出结果可以看到，无论继承了 Thread 类还是实现了 Runnable 接口，运行的结果都是一样的。有些读者可能不理解，为什么实现了 Runnable 接口还需要调用 Thread 类中的 start() 方法才能启动多线程呢？查找 JDK 文档就可以发现，在 Runnable 接口中只有一个 run() 方法，如图所示。

Method Summary	
void	run() When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.
Method Detail	
run	
public void run()	

也就是说在 Runnable 接口中并没有 start() 方法，所以一个类实现了 Runnable 接口也必须用 Thread 类中的 start() 方法来启动多线程。对这一点，通过查找 JDK 文档中的 Thread 类可以看到，在 Thread 类之中有这样一个构造方法：

```
public Thread(Runnable target)
```

由此构造方法可以看到，将一个 Runnable 接口的实例化对象作为参数去实例化 Thread 类对象。在实际的开发中，希望读者尽可能地使用 Runnable 接口去实现多线程机制。

19.2.3 两种多线程实现机制的比较

从前面的分析得知，不管实现了 Runnable 接口还是继承了 Thread 类，其结果都是一样的，那么这两者之间到底有什么关系？

读者可以通过查看 JDK 文档寻找二者之间的联系，如图所示。

java.lang
Class Thread
java.lang.Object └ java.lang.Thread
All Implemented Interfaces: Runnable
public class Thread extends Object implements Runnable

可以看到，Thread 类实现了 Runnable 接口，也就是说 Thread 类也是 Runnable 接口的一个子类。

那么两者之间除了这些联系之外还有什么区别呢？下面通过编写一个应用程序来比较分析。程序 ThreadDemo_3.java 是一个模拟铁路售票系统的范例，实现 4 个售票点发售某日某次列车的车票 20 张，一个售票点用一个线程来表示。

首先用继承 Thread 类来实现这个程序。

【范例 19-4】 Thread 类的继承使用实例 1 (代码 19-4.txt)。

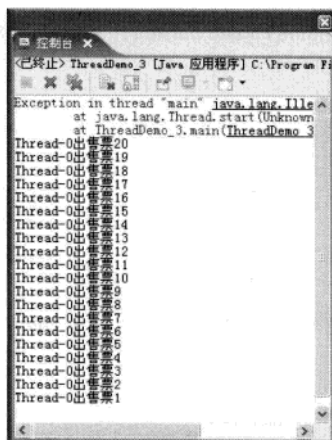
```

01 public class ThreadDemo_3
02 {
03     public static void main(String [] args)
04     {
05         TestThread t=new TestThread();
06         // 一个线程对象只能启动一次
07         t.start();
08         t.start();
09         t.start();
10         t.start();
11     }
12 }
13 class TestThread extends Thread
14 {
15     private int tickets=20;
16     public void run()
17     {
18         while(true)
19         {
20             if(tickets>0)
21                 System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
22         }
23     }
24 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 5 行创建了一个 TestThread 类的实例化对象，之后调用了 4 次 start() 方法。但从运行结果可以看到，程序运行时出现了异常，之后却只有一个线程在运行。这说明了一个类继承 Thread 类之后，这个类的对象无论调用多少次 start() 方法，结果都只有一个线程在运行。

另外，在第 21 行可以看到这样一条语句 “Thread.currentThread().getName()”，此语句表示取得当前运行的线程名称，此方法会在后面讲解。

下面修改 ThreadDemo_3 程序，让 main() 方法中产生 4 个线程。

【范例 19-5】 Thread 类的继承使用实例 2（代码 19-5.txt）。

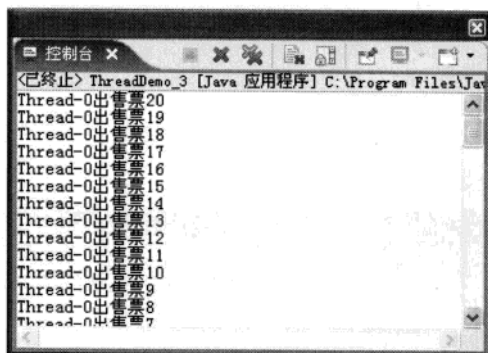
```

01  public class ThreadDemo_3
02  {
03      public static void main(String [] args)
04      {
05          // 启动了 4 个线程，分别进行各自的操作
06          new TestThread().start();
07          new TestThread().start();
08          new TestThread().start();
09          new TestThread().start();
10      }
11  }
12  class TestThread extends Thread
13  {
14      private int tickets=20;
15      public void run()
16      {
17          while(true)
18          {
19              if(tickets>0)
20                  System.out.println(Thread.currentThread().getName()
21                  +"出售票"+tickets--);
22          }
23      }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

由于程序的输出结果过长，所以只截取了后面一部分，但从这部分输出结果中可以看到，这里启动了 4 个线程对象，但这 4 个线程对象各自占有各自的资源，所以可以得出结论：用 Thread 类实际上无法达到资源共享的目的。

那么实现 Runnable 接口会如何呢？下面的这个范例也修改自 ThreadDemo_3，读者可以观察输出的结果。

【范例 19-6】 用 Runnable 接口实现多线程使用实例 2（代码 19-6.txt）。

```

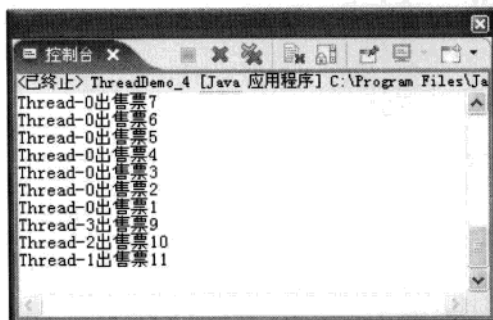
01  public class ThreadDemo_4
02  {
03      public static void main(String [] args)
04      {
05          TestThread t = new TestThread();
06          // 启动了 4 个线程，并实现了资源共享的目的
07          new Thread(t).start();
08          new Thread(t).start();
09          new Thread(t).start();
10          new Thread(t).start();
11      }
12  }
13  class TestThread implements Runnable
14  {
15      private int tickets=20;
16      public void run()
17      {
18          while(true)
19          {
20              if(tickets>0)
21                  System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
22          }

```

```
23     }  
24 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

第 7~10 行启动了 4 个线程。从程序的输出结果来看，尽管启动了 4 个线程对象，但结果都是操纵同一个资源，从而实现了资源共享的目的。

可见，实现 Runnable 接口相对于继承 Thread 类来说，有如下几个显著的优势。

(1) 适合多个相同程序代码的线程去处理同一资源的情况，把虚拟 CPU（线程）同程序的代码、数据有效分离，较好地体现了面向对象的设计思想。

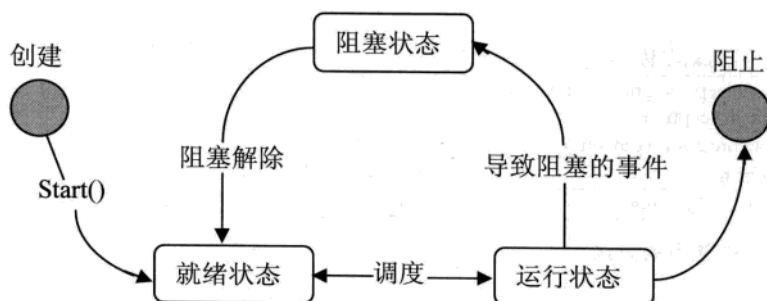
(2) 可以避免由于 Java 的单继承特性带来的局限。开发中经常会碰到这样一种情况，即要将已经继承了某一个类的子类放入多线程中，由于一个类不能同时有两个父类，所以不能使用继承 Thread 类的方式，那么就只能采用实现 Runnable 接口的方式。

(3) 增强了程序的健壮性，代码能够被多个线程共享，代码与数据是独立的。当多个线程的执行代码来自同一个类的实例时，即称它们共享相同的代码。多个线程可以操作相同的数据，与它们的代码无关。当共享访问相同的对象时，即共享相同的数据。当线程被构造时，需要的代码和数据通过一个对象作为构造方法实参传递进去，这个对象就是一个实现了 Runnable 接口的类的实例。

事实上，几乎所有的多线程应用都可以使用第 2 种方式，即实现 Runnable 接口。

19.3 线程的状态

每个 Java 程序都有一个默认的主线程，对于 Java 应用程序，主线程是 main() 方法执行的线索；对于 Applet 程序，主线程是指浏览器加载并执行 Java Applet 程序的线索。要想实现多线程，必须在主线程中创建新的线程对象。任何一个线程一般都具有 5 种状态，即创建、就绪、运行、阻塞、终止。线程状态的转移与方法之间的关系如图所示。



在给定时间点上，一个线程只能处于一种状态。

- (1) NEW：至今尚未启动的线程处于这种状态。
- (2) RUNNABLE：正在 Java 虚拟机中执行的线程处于这种状态。
- (3) BLOCKED：受阻塞并等待某个监视器锁的线程处于这种状态。
- (4) WAITING：无限期地等待另一个线程来执行某一特定操作的线程处于这种状态。
- (5) TIMED_WAITING：等待另一个线程来执行取决于指定等待时间的操作的线程处于这种状态。
- (6) TERMINATED：已退出的线程处于这种状态。

19.4 线程操作的一些方法

本节视频教学录像：30 分钟

操作线程的主要方法在 Thread 类中，下表列出了 Thread 类中的主要方法。

方法名称	方法说明
public static int activeCount()	返回线程组中目前活动的线程的数目
public static native Thread currentThread()	返回目前正在执行的线程
public void destroy()	销毁线程
public static int enumerate(Thread tarray[])	将当前和子线程组中的活动线程复制至指定的线程数组
public final String getName()	返回线程的名称
public final int getPriority()	返回线程的优先级
public final ThreadGroup getThreadGroup()	返回线程的线程组
public static boolean interrupted()	判断目前线程是否被中断，如果是，返回 true；否则返回 false
public final native boolean isAlive()	判断线程是否在活动，如果是，返回 true；否则返回 false
public boolean isInterrupted()	判断目前线程是否被中断，如果是，返回 true；否则返回 false
public final void join() throws InterruptedException	等待线程死亡
public final synchronized void join(long millis) throws InterruptedException	等待 millis 毫秒后，线程死亡
public final synchronized void join(long millis, int nanos) throws InterruptedException	等待 millis 毫秒加上 nanos 微秒后，线程死亡
public void run()	执行线程
public final void setName()	设定线程名称
public final void setPriority(int newPriority)	设定线程的优先值
public static native void sleep(long millis) throws InterruptedException	使目前正在执行的线程休眠 millis 毫秒

续表

方法名称	方法说明
public static void sleep(long millis,int nanos) throws InterruptedException	使目前正在执行的线程休眠 millis 毫秒加上 nanos 微秒
public native synchronized void start()	开始执行线程
public String toString()	返回代表线程的字符串
public static native void yield()	将目前正在执行的线程暂停，允许其他线程执行

下面介绍一些常用的方法。

19.4.1 取得和设置线程的名称

在 Thread 类中，可以通过 getName()方法取得线程的名称，通过 setName()方法设置线程的名称。

线程的名称一般在启动线程前设置，但也允许为已经运行的线程设置名称。允许两个 Thread 对象有相同的名字，但为了清晰，应该尽量避免这种情况的发生。

另外，如果程序并没有为线程指定名称，系统会自动为线程分配名称，如下面范例所示。

【范例 19-7】 线程名称的分配（代码 19-7.txt）。

```

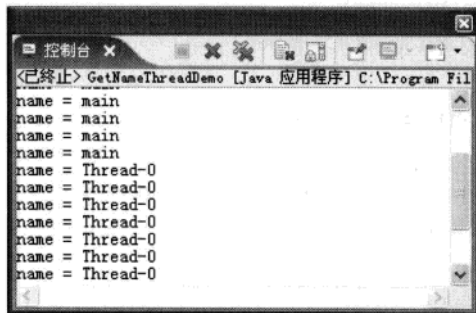
01 public class GetNameThreadDemo extends Thread
02 {
03     public void run()
04     {
05         for(int i=0;i<10;i++)
06             printMsg();
07     }
08     public void printMsg()
09     {
10         // 获得运行此代码的线程的引用
11         Thread t = Thread.currentThread();
12         String name = t.getName();
13         System.out.println("name = "+name);
14     }
15     public static void main(String[] args)
16     {
17         GetNameThreadDemo t1 = new GetNameThreadDemo();
18         t1.start();
19         for(int i=0;i<10;i++)
20         {
21             t1.printMsg();
22         }
23     }

```

24 }

【运行结果】

保存并运行程序，结果如图所示。

**【代码详解】**

第 1 行声明了一个 GetNameThreadDemo 类，此类继承自 Thread 类，之后第 3~7 行覆写 Thread 类中的 run() 方法。

第 8~14 行声明了一个 printMsg() 方法，此方法用于取得当前线程的信息。在第 11 行，通过 Thread 类中的 currentThread() 方法，返回一个 Thread 类的实例化对象，在表中可以看到，此方法返回当前正在运行的线程，即返回正在调用此方法的线程。第 12 行通过调用 Thread 类中的 getName() 方法，返回当前运行的线程的名称。

第 6 行和第 21 行分别调用了 printMsg() 方法，但第 6 行是从多线程的 run() 方法中调用，而第 21 行则是从 main() 方法中调用。



提示：有些读者可能不理解，为什么程序中输出的运行线程的名称中会有一个 main 呢？这是因为 main() 方法也是一个线程，实际上在命令行中运行 java 命令时，就启动了一个 JVM 的进程，默认情况下此进程会产生两个线程：一个是 main() 方法线程，另外一个就是垃圾回收（GC）线程。

下面看一下如何在线程中设置线程的名称。

【范例 19-8】 在线程中设置线程的名称（代码 19-8.txt）。

```

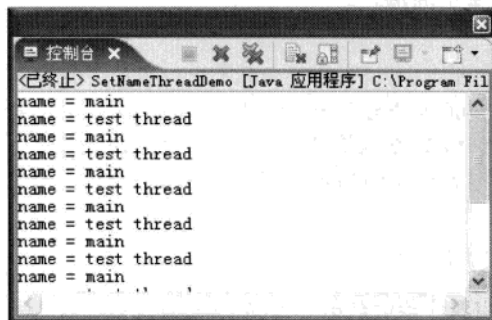
01 public class SetNameThreadDemo extends Thread
02 {
03     public void run()
04     {
05         for(int i=0;i<10;i++)
06         {
07             printMsg();
08         }
09     }

```

```
10     public void printMsg()
11     {
12         // 获得运行此代码的线程的引用
13         Thread t = Thread.currentThread();
14         String name = t.getName();
15         System.out.println("name = "+name);
16     }
17     public static void main(String args[])
18     {
19         SetNameThreadDemo tt = new SetNameThreadDemo();
20         // 在这里设置线程的名称
21         tt.setName("test thread");
22         tt.start();
23         for(int i=0;i<10;i++)
24         {
25             tt.printMsg();
26         }
27     }
28 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

本程序与上面的程序类似，唯一的不同之处在于第 21 行调用了 Thread 类中的 setName() 方法，用于设置线程的名称，所以从运行结果中可以看到，运行的线程中有一个名称为“test thread”的线程。

19.4.2 判断线程是否启动

在程序中也可以通过 isAlive()方法来测试线程是否已经启动而且仍然在启动。

【范例 19-9】 判断线程是否启动（代码 19-9.txt）。

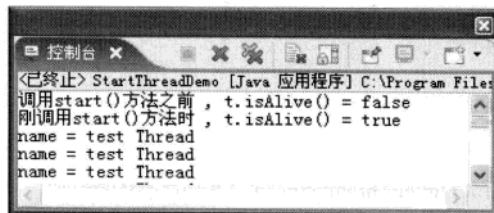
```

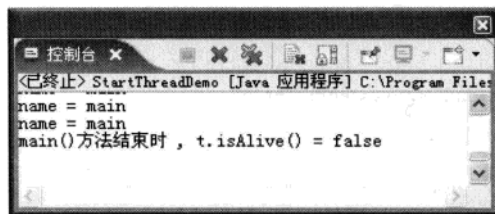
01 public class StartThreadDemo extends Thread
02 {
03     public void run()
04     {
05         for(int i=0;i<10;i++)
06         {
07             printMsg();
08         }
09     }
10     public void printMsg()
11     {
12         // 获得运行此代码的线程的引用
13         Thread t = Thread.currentThread();
14         String name = t.getName();
15         System.out.println("name = "+name);
16     }
17     public static void main(String[] args) {
18         StartThreadDemo t = new StartThreadDemo();
19         t.setName("test Thread");
20         System.out.println("调用 start()方法之前 , t.isAlive() = "+t.isAlive());
21         t.start();
22         System.out.println("刚调用 start()方法时 , t.isAlive() = "+t.isAlive());
23         for(int i=0;i<10;i++)
24         {
25             t.printMsg();
26         }
27         // 下面语句的输出结果是不固定的, 有时输出 false, 有时输出 true
28         System.out.println("main()方法结束时 , t.isAlive() = "+t.isAlive());
29     }
30 }

```

【运行结果】

保存并运行程序，结果如图所示。





【代码详解】

第 20 行在线程运行之前调用 `isAlive()` 方法，判断线程是否启动，但在此处并没有启动，所以返回 “false”，表示线程未启动。

第 22 行在启动线程之后调用 `isAlive()` 方法，此时线程已经启动，所以返回 “true”。

第 28 行在 `main()` 方法快结束时调用 `isAlive()` 方法，此时的状态不再固定，有可能是 true，有可能是 false。

19.4.3 后台线程与 `setDaemon()` 方法

对 Java 程序来说，只要还有一个前台线程在运行，这个进程就不会结束，如果一个进程中只有后台线程在运行，这个进程就会结束。前台线程是相对后台线程而言的，前面所介绍的线程都是前台线程。那么什么样的线程是后台线程呢？如果某个线程对象在启动（调用 `start()` 方法）之前调用了 `setDaemon(true)` 方法，这个线程就变成了后台线程。下面看一下进程中只有后台线程在运行的情况。

【范例 19-10】 `setDaemon()` 方法的使用（代码 19-10.txt）。

```

01 public class ThreadDaemon
02 {
03     public static void main(String args[])
04     {
05         ThreadTest t = new ThreadTest();
06         Thread tt = new Thread(t);
07         tt.setDaemon(true);           // 设置后台运行
08         tt.start();
09     }
10 }
11
12 class ThreadTest implements Runnable
13 {
14     public void run()
15     {
16         while(true)
17         {
18             System.out.println(Thread.currentThread().getName()+"is running.");

```

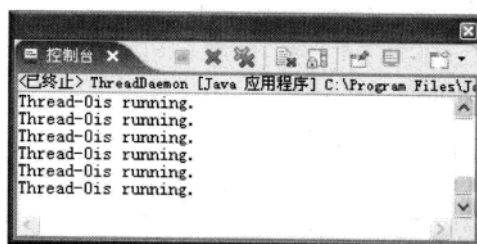
```

19         }
20     }
21 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

从程序和运行结果图中可以看到：虽然创建了一个无限循环的线程，但因为它是后台线程，因此整个进程在主线程结束时就随之终止运行了。这验证了进程中只有后台线程运行时，进程就会结束的说法。

19.4.4 线程的强制运行

在讲解此概念之前，请先看下面的范例。

【范例 19-11】 线程的强制运行（代码 19-11.txt）。

```

01  public class ThreadJoin
02  {
03      public static void main(String[] args)
04      {
05          ThreadTest t=new ThreadTest();
06          Thread pp=new Thread(t);
07          pp.start();
08          int i=0;
09          for(int x=0;x<10;x++)
10          {
11              if(i==5)
12              {
13                  try
14                  {
15                      pp.join();           // 强制运行完一线程后，再运行后面的线程
16                  }
17                  catch(Exception e)      // 会抛出 InterruptedException

```

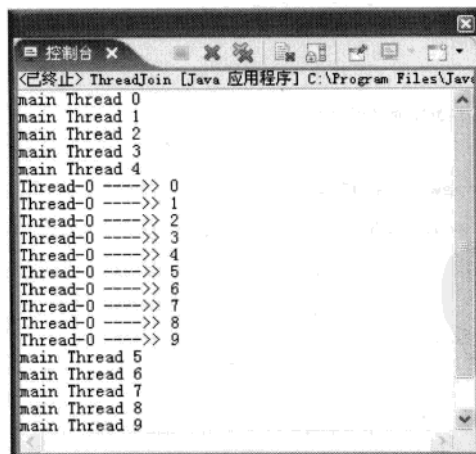
```

18         {
19             System.out.println(e.getMessage());
20         }
21     }
22     System.out.println("main Thread "+i++);
23 }
24 }
25 }
26 class ThreadTest implements Runnable
27 {
28     public void run()
29     {
30         String str=new String();
31         int i=0;
32         for(int x=0;x<10;x++)
33         {
34             System.out.println(Thread.currentThread().getName()+" ---->> "+i++);
35         }
36     }
37 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

本程序启动了两个线程，一个是 main() 线程，另一个是 pp 线程。

第 15 行调用了 pp 线程对象的 join() 方法，在程序的输出结果中可以看到，调用 join() 方法之后，只有 pp 的线程对象在运行，也就是说，join() 方法是用来强制某一线程的运行。

【范例分析】

由上可见，pp 线程中的代码被并入到了 main 线程中，也就是 pp 线程中的代码不执行完，main 线程中的代码就只能一直等待。查看 JDK 文档可以发现，除了无参数的 join 方法外，还有两个带参数的 join 方法，分别是 join(long millis)和 join(long millis,int nanos)，它们的作用是指定合并时间，前者精确到毫秒，后者精确到纳秒，意思是两个线程合并指定的时间后，又开始分离，回到合并前的状态。读者可以把上面的程序中的 join 方法修改成为有参数的，再看看程序运行的结果。

19.4.5 线程的休眠

在 Thread 类中有一个名为 sleep(long millis)的静态方法，此方法用于线程的休眠。

【范例 19-12】 线程的休眠（代码 19-12.txt）。

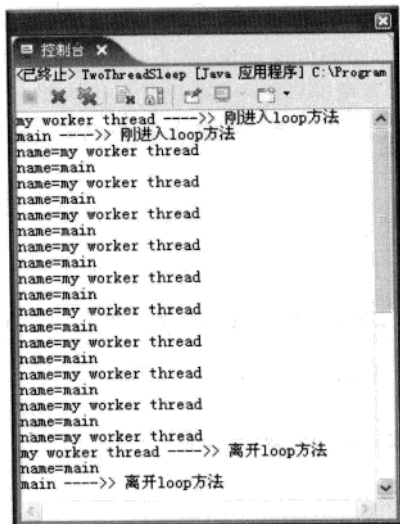
```

01 public class TwoThreadSleep extends Thread {
02     public void run() {
03         loop();
04     }
05     public void loop() {
06         String name = Thread.currentThread().getName();
07         System.out.println(name+" ---->> 刚进入 loop 方法");
08         for ( int i = 0; i < 10; i++ )
09             {
10                 try {
11                     Thread.sleep(2000);
12                 } catch ( InterruptedException x ) {}
13                 System.out.println("name="+ name);
14             }
15         System.out.println(name+" ---->> 离开 loop 方法");
16     }
17     public static void main(String[] args)
18     {
19         TwoThreadSleep tt = new TwoThreadSleep();
20         tt.setName("my worker thread");
21         tt.start();
22         try {
23             Thread.sleep(700);
24         } catch ( InterruptedException x ) {}
25         tt.loop();
26     }
27 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 11 行和第 23 行分别使用了 `sleep()` 方法，所以运行此程序时会发现运行的速度明显降低了很多，这是因为每次运行时都需要先休眠一会儿。由于使用 `sleep()` 方法会抛出一个 `InterruptedException`，所以在程序中需要用 `try...catch()` 捕获。

19.4.6 线程的中断

当一个线程运行时，另一个线程可以调用对应的 `Thread` 对象的 `interrupt()` 方法来中断它。

【范例 19-13】 线程的中断使用范例 1（代码 19-13.txt）。

```
01 public class SleepInterrupt implements Runnable
02 {
03     public void run()
04     {
05         try {
06             System.out.println("在 run()方法中 - 这个线程休眠 20 秒");
07             Thread.sleep(20000);
08             System.out.println("在 run()方法中 - 继续运行");
09         }
10         catch (InterruptedException x) {
11             System.out.println("在 run()方法中 - 中断线程");
12             return;
13         }
14     }
15 }
```



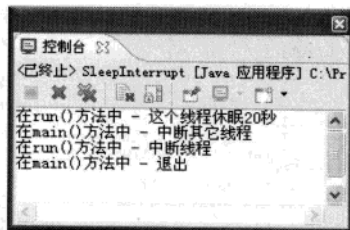
```

14      System.out.println("在 run()方法中 - 休眠之后继续完成");
15      System.out.println("在 run()方法中 - 正常退出");
16  }
17  public static void main(String[] args)
18  {
19      SleepInterrupt si = new SleepInterrupt();
20      Thread t = new Thread(si);
21      t.start();
22      // 在此休眠是为确保线程能运行一会
23      try {
24          Thread.sleep(2000);
25      }
26      catch (InterruptedException x) {}
27      System.out.println("在 main()方法中 - 中断其他线程");
28      t.interrupt();
29      System.out.println("在 main()方法中 - 退出");
30  }
31  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

SleepInterrupt 类实现了 Runnable 接口，同时覆写了 run() 方法，在 run() 方法之中将线程休眠 20 秒。

第 21 行调用 start() 方法，此方法用于启动线程。

第 23~26 行调用 sleep() 方法，将线程休眠 2 秒，这样做是为了保证 run() 方法中的内容能够多执行一会儿。

第 28 行，因为 Thread 对象 t 在 main() 方法中，所以由 main() 线程调用 interrupt() 方法，将另外一个线程中断。

也可以用 Thread 对象调用 isInterrupted() 方法来检查每个线程的中断状态。

【范例 19-14】 线程的中断使用范例 2（代码 19-14.txt）。

```
01 public class InterruptCheck
```

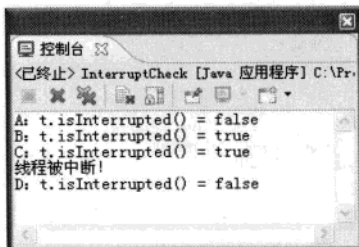
```

02  {
03      public static void main(String[] args)
04      {
05          Thread t = Thread.currentThread();
06          System.out.println("A: t.isInterrupted() = " + t.isInterrupted());
07          t.interrupt();
08          System.out.println("B: t.isInterrupted() = " + t.isInterrupted());
09          System.out.println("C: t.isInterrupted() = " + t.isInterrupted());
10          try {
11              Thread.sleep(2000);
12              System.out.println("线程没有被中断!");
13          } catch (InterruptedException x) {
14              System.out.println("线程被中断!");
15          }
16          // 因为 sleep 抛出了异常, 所以它清除了中断标志
17          System.out.println("D: t.isInterrupted() = " + t.isInterrupted());
18      }
19  }

```

【运行结果】

保存并运行程序, 结果如图所示。



【代码注解】

第5行通过 Thread 类中的 currentThread() 方法取得当前运行的线程, 因为此代码是在 main() 方法中运行, 所以当前的线程就为 main() 线程。

第6行因为没有调用中断方法, 所以此时线程未中断。但在第7行调用了中断方法, 所以之后的线程状态都为中断。

第11行让线程开始休眠, 但此时线程已经被中断, 所以这个时候会抛出中断异常, 抛出中断异常之后会清除中断标记, 所以最后在判断是否中断的时候, 会返回线程未中断。

19.5 多线程的同步

▶ 本节视频教学录像: 47 分钟

本节介绍多线程的同步, 具体介绍同步问题的引出、同步代码块、同步方法和死锁等内容。

19.5.1 同步问题的引出

在前面讲解过的卖票程序中，极有可能碰到一种意外，就是同一张票号被打印两次或多次，也可能出现打印出的票号为 0 或是负数的情况。这个意外出现的原因出现在下面的这部分代码中。

```
if(tickets>0)
    System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
```

假设 tickets 的值为 1 的时候，线程 1 刚执行完 if(tickets>0) 这行代码，正准备执行下面的代码，就在这时，操作系统将 CPU 切换到了线程 2 上执行，此时 tickets 的值仍为 1，线程 2 执行完上面两行代码，tickets 的值变为 0 后，CPU 又切回到了线程 1 上执行，但此时线程 1 不会再执行 if(tickets>0) 这行代码，因为先前已经比较过了，并且比较的结果为真，线程 1 将直接往下执行这行代码。

```
System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
```

但此刻 tickets 的值已变为 0，屏幕打印出来的将是 0。

要想立即见到这种意外，可以在程序中调用 Thread.sleep() 静态方法来刻意造成线程间的这种切换。Thread.sleep() 方法将迫使线程执行到该处后暂停执行，让出 CPU 给别的线程，在指定的时间（这里是毫秒）后，CPU 回到刚才暂停的线程上执行。修改完的 TestThread 代码如下。

【范例 19-15】 线程的同步问题（代码 19-15.txt）。

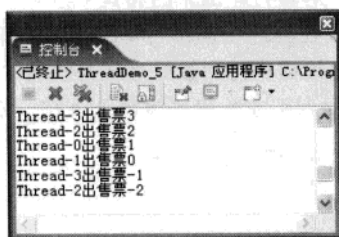
```
01 public class ThreadDemo_5
02 {
03     public static void main(String [] args)
04     {
05         TestThread t = new TestThread();
06         // 启动了 4 个线程，实现了资源共享的目的
07         new Thread(t).start();
08         new Thread(t).start();
09         new Thread(t).start();
10         new Thread(t).start();
11     }
12 }
13 class TestThread implements Runnable
14 {
15     private int tickets=20;
16     public void run()
17     {
18         while(true)
19         {
```

```

20         if(tickets>0)
21         {
22             try{
23                 Thread.sleep(100);
24             }
25             catch(Exception e){}
26             System.out.println(Thread.currentThread().getName()
                + "出售票"+tickets--);
27         }
28     }
29 }
30 }
    
```

【运行结果】

在本程序中，故意实现线程执行完 `if(tickets>0)` 语句后，执行 `Thread.sleep(100)`，以让出 CPU 给别的线程。编译运行程序，屏幕上出现的最后几行结果如图所示。



【范例分析】

从运行结果可以看到，票号被打印出来了负数，这说明有同一张票被卖了 3 次的意外发生。

造成这种意外的根本原因就是资源数据访问不同步引起的。那么该如何去解决这个问题呢？为此下面引入同步的概念。

19.5.2 同步代码块

如何避免上面的这种意外出现呢？如何保证开发出的程序是线程安全的呢？这就要涉及线程间的同步问题。要解决上面的问题，必须保证下面这段代码的原子性。

```

if(tickets>0)
{
    System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
}
    
```

即当一个线程运行到 `if(tickets>0)` 后，CPU 不去执行其他线程中的、可能影响当前线程中的下一句代码的执行结果的代码块，必须等到下一句执行完后才能去执行其他线程中的有关代码块。这段代码就好比一座独木桥，任何时刻都只能有一个人在桥上行走，即程序中不能有多个线

程同时在这两句代码之间执行，这就是线程同步。同步代码块定义语法如下。

```
...
synchronized(对象)
{
    需要同步的代码 ;
}
...
```

现在修改 ThreadDemo_5 程序中的 TestThread 类，使程序具有同步性，修改后的代码如下。

```
class TestThread implements Runnable
{
    private int tickets=20;
    public void run()
    {
        while(true)
        {
            synchronized(this)
            {
                if(tickets>0)
                {
                    try{
                        Thread.sleep(100);
                    }
                    catch(Exception e){}
                    System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
                }
            }
        }
    }
}
```

本程序将这些需要具有原子性的代码放入 synchronized 语句内，形成了同步代码块。在同一时刻只能有一个线程可以进入同步代码块内运行，只有当该线程离开同步代码块后，其他线程才能进入同步代码块内运行。

19.5.3 同步方法

除了可以对代码块进行同步外，也可以对方法实现同步，只要在需要同步的方法定义前面加上 synchronized 关键字即可。同步方法定义语法如下。

访问控制符 synchronized 返回值类型 方法名称(参数)


```
{  
    ....;  
}
```

根据上述格式，修改 TestThread 类。

```
01 class TestThread implements Runnable  
02 {  
03     private int tickets=20;  
04     public void run()  
05     {  
06         while(true)  
07         {  
08             sale();  
09         }  
10     }  
11     public synchronized void sale()  
12     {  
13         if(tickets>0)  
14         {  
15             try{  
16                 Thread.sleep(100);  
17             }  
18             catch(Exception e){}  
19             System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);  
20         }  
21     }  
22 }
```

可见，编译运行后的结果同上面同步代码块方式的运行结果完全一样，也就是说在方法定义前使用 synchronized 关键字也能够很好地实现线程间的同步。

在同一类中，使用 synchronized 关键字定义的若干方法，可以在多个线程之间同步。当有一个线程进入了有 synchronized 修饰的方法时，其他线程就不能进入同一个对象使用 synchronized 来修饰所有方法，直到第 1 个线程执行完它所进入的 synchronized 修饰的方法为止。

19.5.4 死锁

一旦有多个进程，且它们都要争用对多个锁的独占访问，那么就有可能发生死锁。如果有一组进程或线程，其中每个都在等待一个只有其他进程或线程才可以进行的操作，那么就称它们被死锁了。

最常见的死锁形式是当线程 1 持有对象 A 上的锁，而且正在等待对象 B 上的锁；而线程

2 持有对象 B 上的锁，却正在等待对象 A 上的锁。这两个线程永远都不会获得第 2 个锁，或是释放第 1 个锁，所以它们只会永远等待下去。这就好比两个人在吃饭，甲拿到了一根筷子和一把刀子，乙拿到了一把叉子和一根筷子，他们都无法吃到饭。

于是，就发生下面的事件。

甲：“你先给我筷子，我再给你刀子！”

乙：“你先给我刀子，我才给你筷子！”

.....

结果可想而知，谁也没法吃到饭。

要避免死锁，应该确保在获取多个锁时，在所有的线程中都以相同的顺序获取锁。

在下面的例子中，程序创建了两个类 A 和 B，它们分别具有方法 funA() 和 funB()，在调用对方的方法前，funA() 和 funB() 都睡眠一会儿。主类 DeadLockDemo 创建 A 和 B 实例，然后产生第 2 个线程以构成死锁条件。funA() 和 funB() 使用 sleep() 方法来强制死锁条件出现。而在真实程序中，死锁是较难发现的。

【范例 19-16】 程序死锁的产生（代码 19-16.txt）。

```

01  class A
02  {
03      synchronized void funA(B b)
04      {
05          String name=Thread.currentThread().getName();
06          System.out.println(name+" 进入 A.foo ");
07          try
08          {
09              Thread.sleep(1000);
10          }
11          catch(Exception e)
12          {
13              System.out.println(e.getMessage());
14          }
15          System.out.println(name+" 调用 B 类中的 last()方法");
16          b.last();
17      }
18      synchronized void last()
19      {
20          System.out.println("A 类中的 last()方法");
21      }
22  }
23  class B
24  {
25      synchronized void funB(A a)

```

```

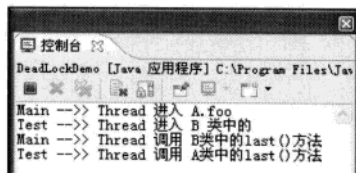
26     {
27         String name=Thread.currentThread().getName();
28         System.out.println(name + " 进入 B 类中的");
29         try
30         {
31             Thread.sleep(1000);
32         }
33         catch(Exception e)
34         {
35             System.out.println(e.getMessage());
36         }
37         System.out.println(name + " 调用 A 类中的 last()方法");
38         a.last();
39     }
40     synchronized void last()
41     {
42         System.out.println("B 类中的 last()方法");
43     }
44 }
45 class DeadLockDemo implements Runnable
46 {
47     A a=new A();
48     B b=new B();
49     DeadLockDemo()
50     {
51         // 设置当前线程的名称
52         Thread.currentThread().setName("Main -->> Thread");
53         new Thread(this).start();
54         a.funA(b);
55         System.out.println("main 线程运行完毕");
56     }
57     public void run()
58     {
59         Thread.currentThread().setName("Test -->> Thread");
60         b.funB(a);
61         System.out.println("其他线程运行完毕");
62     }
63     public static void main(String[] args)
64     {
65         new DeadLockDemo();
66     }

```

67 }

【运行结果】

保存并运行程序，结果如图所示。

**【范例分析】**

从运行结果可以看到，Test --> Thread 进入了 b 的监视器，然后又在等待 a 的监视器。同时 Main --> Thread 进入了 a 的监视器，并等待 b 的监视器。这个程序永远不会完成。

19.6 线程间通信

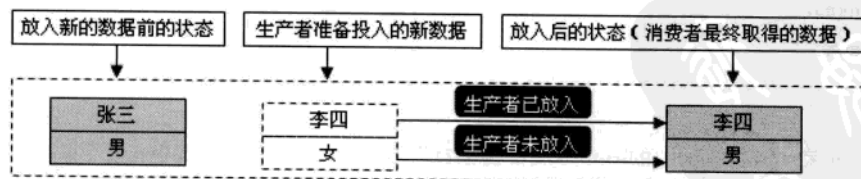
本节介绍线程间通信，具体介绍问题的引出和问题如何解决等内容。

19.6.1 问题的引出

下面通过这样的应用来讲解线程间的通信。把一个数据存储空间划分为两部分：一部分用于存储人的姓名，另一部分用于存储人的性别。这里的应用包含两个线程：一个线程向数据存储空间添加数据（生产者），另一个线程从数据存储空间中取出数据（消费者）。这个程序有两种意外需要读者考虑。

第 1 种意外：假设生产者线程刚向数据存储空间中添加了一个人的姓名，还没有加入这个人的性别，CPU 就切换到了消费者线程，消费者线程则把这个人的姓名和上一个人的性别联系到了一起。

这个过程可用下图表示。



第 2 种意外：生产者放入了若干次数据，消费者才开始取数据，或者是，消费者取完一个数据后，还没等到生产者放入新的数据，又重复取出已取过的数据。

19.6.2 问题如何解决

下面先来构思这个程序，程序中的生产者线程和消费者线程运行的是不同的程序代码，因此

这里需要编写两个包含有 run 方法的类来完成这两个线程，一个是生产者类 Producer，另一个是消费者类 Consumer。

```
class Producer implements Runnable
{
    public void run()
    {
        while(true)
        {
            // 编写往数据存储空间中放入数据的代码
        }
    }
}
```

```
class Consumer implements Runnable
{
    public void run()
    {
        while(true)
        {
            // 编写从数据存储空间中读取数据的代码
        }
    }
}
```

当程序写到这里，还需要定义一个新的数据结构来作为数据存储空间。

```
class P
{
    String name;
    String sex;
}
```

Producer 和 Consumer 中的 run 方法都需要操作类 P 的同一个对象实例。接下来，对 Producer 和 Consumer 这两个类作如下修改，顺便写出程序的主调用类 ThreadCommuation。

【范例 19-17】 线程之间的通信（代码 19-17.txt）。

```
01 class Producer implements Runnable
02 {
03     P q=null;
```

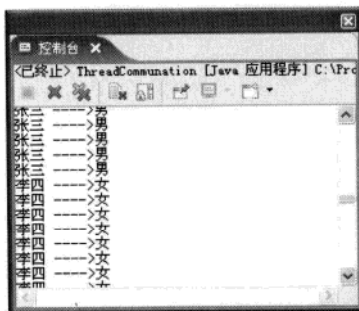


```
04     public Producer(P q)
05     {
06         this.q=q;
07     }
08     public void run()
09     {
10         int i=0;
11         while(true)
12         {
13             if(i==0)
14             {
15                 q.name="张三";
16                 q.sex="男";
17             }
18             else
19             {
20                 q.name="李四";
21                 q.sex="女";
22             }
23             i=(i+1)%2;
24         }
25     }
26 }
27 class P
28 {
29     String name="李四";
30     String sex="女";
31 }
32 class Consumer implements Runnable
33 {
34     P q=null;
35     public Consumer(P q)
36     {
37         this.q=q;
38     }
39     public void run()
40     {
41         while(true)
42         {
43             System.out.println(q.name + " ---->" + q.sex);
44         }
```

```
45     }
46 }
47 public class ThreadCommuation
48 {
49     public static void main(String [] args)
50     {
51         P q=new P();
52         new Thread(new Producer(q)).start();
53         new Thread(new Consumer(q)).start();
54     }
55 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

从输出结果可以看到，原本李四是女、张三是男，现在却打印出了张三是女的奇怪现象，这是什么原因？从程序中可以看到，Producer类和Consumer类都是操纵了同一个P类，这就有可能Producer类还未操纵完P类，Consumer类就已经将P类中的内容取走了，这就是资源不同步的原因。为此可以在P类中增加两个同步方法：set()和get()，具体代码如下所示。

【范例 19-18】 进程同步使用（代码 19-18.txt）。

```
01 class Producer implements Runnable
02 {
03     P q=null;
04     public Producer(P q)
05     {
06         this.q=q;
07     }
08     public void run()
09     {
10         int i=0;
```

```
11     while(true)
12     {
13         if(i==0)
14         {
15             q.set("张三","男");
16         }
17         else
18         {
19             q.set("李四","女");
20         }
21         i=(i+1)%2;
22     }
23 }
24 }
25 class P
26 {
27     private String name="李四";
28     private String sex="女";
29     public synchronized void set(String name,String sex)
30     {
31         this.name = name ;
32         this.sex =sex ;
33     }
34     public synchronized void get()
35     {
36         System.out.println(this.name + "---->" + this.sex );
37     }
38 }
39 class Consumer implements Runnable
40 {
41     P q=null;
42     public Consumer(P q)
43     {
44         this.q=q;
45     }
46     public void run()
47     {
48         while(true)
49         {
50             q.get();
51         }
```

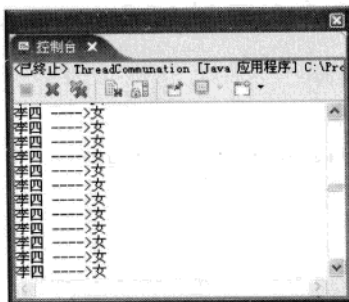
```

52     }
53 }
54 public class ThreadCommuation
55 {
56     public static void main(String [] args)
57     {
58         P q=new P();
59         new Thread(new Producer(q)).start();
60         new Thread(new Consumer(q)).start();
61     }
62 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

可以看到程序的输出结果是正确的。但是这里又有一个新的问题产生了，从程序的执行结果来看，Consumer 线程对 Producer 线程放入的一次数据连续地读取了多次，这并不符合实际的要求。实际要求的结果是，Producer 放一次数据，Consumer 就取一次；反之，Producer 也必须等到 Consumer 取完后才能放入新的数据，而这一问题的解决就需要使用下面要讲到的线程间的通信。Java 是通过 Object 类的 wait、notify、notifyAll 这几个方法来实现线程间的通信的，又因为所有的类都是从 Object 继承的，所以任何类都可以直接使用这些方法。

下面是这 3 个方法的简要说明。

wait: 告诉当前线程放弃监视器并进入睡眠状态，直到其他线程进入同一监视器并调用 notify 为止。

notify: 唤醒同一对象监视器中调用 wait 的第 1 个线程。这类似排队买票，一个人买完之后，后面的人才可以继续买。

notifyAll: 唤醒同一对象监视器中调用 wait 的所有线程，具有最高优先级的线程首先被唤醒并执行。

如果能让上面的程序符合预先的设计需求，就必须在类 P 中定义一个新的成员变量 bFull 来表示数据存储空间的状态。当 Consumer 线程取走数据后，bFull 值为 false，当 Producer 线程放入数据后，bFull 值为 true。只有 bFull 为 true 时，Consumer 线程才能取走数据，否则就必须等

待 Producer 线程放入新的数据后的通知；反之，只有 bFull 为 false，Producer 线程才能放入新的数据，否则就必须等待 Consumer 线程取走数据后的通知。修改后的 P 类的程序代码如下。

【范例 19-19】 线程间通信问题的解决（代码 19-19.txt）。

```

01  class P
02  {
03      private String name="李四";
04      private String sex="女";
05      boolean bFull = false ;
06      public synchronized void set(String name,String sex)
07      {
08          if(bFull)
09          {
10              try
11              {
12                  wait();                // 后来的线程要等待
13              }
14              catch(InterruptedException e)
15              {}
16          }
17          this.name = name ;
18          try
19          {
20              Thread.sleep(10);
21          }
22          catch(Exception e)
23          {
24              System.out.println(e.getMessage());
25          }
26          this.sex = sex ;
27          bFull = true ;
28          notify()                        // 唤醒最先到达的线程
29      }
30      public synchronized void get()
31      {
32          if(!bFull)
33          {
34              try
35              {
36                  wait();

```



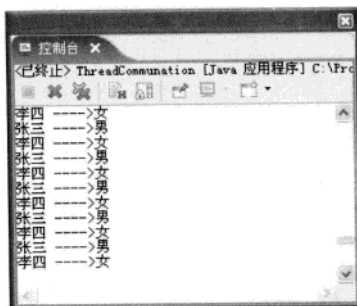
```

37         }
38         catch (InterruptedException e)
39         {}
40     }
41     System.out.println(name+" ---->" +sex);
42     bFull = false ;
43     notify();
44 }
45 }

```

【运行结果】

保存并运行程序，结果如图所示。

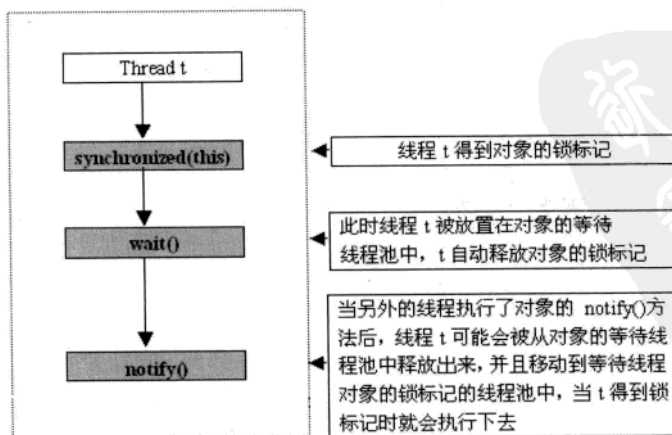


【范例分析】

本程序满足了设计的需求，解决了线程间通信的问题。

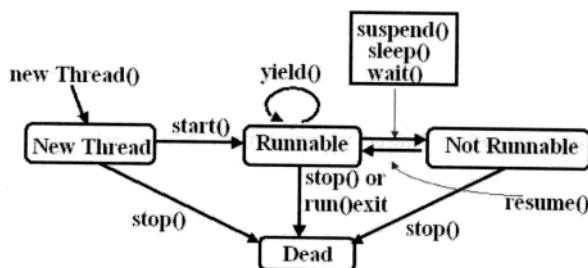
wait、notify、notifyAll 这 3 个方法只能在 synchronized 方法中调用，即无论线程调用一个对象的 wait 还是 notify 方法，该线程必须先得到该对象的锁标记。这样，notify 就只能唤醒同一对象监视器中调用 wait 的线程。而使用多个对象监视器，就可以分别有多个 wait、notify 的情况，同组里的 wait 只能被同组的 notify 唤醒。

一个线程的等待和唤醒过程可以用下图表示。



19.7 线程生命周期的控制

任何事物都有一个生命周期，线程也不例外。那么在一个程序中，怎样控制一个线程的生命并让它更有效地工作呢？要想控制线程的生命，先得了解线程产生和消亡的整个过程。请读者结合前面讲的内容，仔细看一看下面的图。



了解了线程的生命周期，就不难想出能够控制线程生命的办法了吧？其实，控制线程生命周期的方法有多种，如 `suspend` 方法、`resume` 方法和 `stop` 方法。但对这 3 个方法都不推荐使用，其中，不推荐使用 `suspend` 方法和 `resume` 方法的原因如下。

(1) 会导致死锁的发生。

(2) 它允许一个线程(甲)通过直接控制另外一个线程(乙)的代码来直接控制那个线程(乙)。

虽然 `stop` 能够避免死锁的发生，但是也有其不足之处：如果一个线程正在操作共享数据段，操作过程没有完成就被“`stop`”了的话，将会导致数据的不完整性。因此对 `stop` 方法也不推荐使用。

既然对这 3 个方法都不推荐使用，那么到底该使用什么方法？请看下面的代码。

【范例 19-20】 线程的生命周期（代码 19-20.txt）。

```

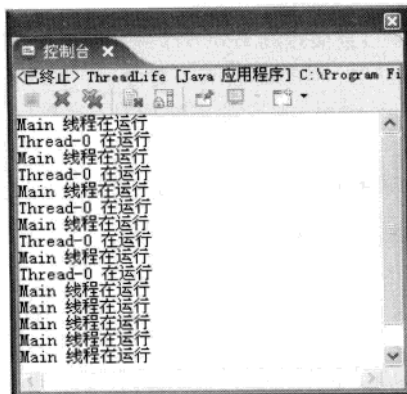
01 public class ThreadLife
02 {
03     public static void main(String [] args)
04     {
05         TestThread t=new TestThread();
06         new Thread(t).start();
07         for(int i=0;i<10; i++)
08         {
09             if(i == 5)
10                 t.stopMe();
11             System.out.println("Main 线程在运行");
12         }
13     }
14 }
15 class TestThread implements Runnable

```

```
16 {  
17     private boolean bFlag = true;  
18     public void stopMe()  
19     {  
20         bFlag = false;  
21     }  
22     public void run()  
23     {  
24         while(bFlag)  
25         {  
26             System.out.println(Thread.currentThread().getName()+" 在运行");  
27         }  
28     }  
29 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

本程序中定义了一个计数器 *i*，用来控制 *main* 线程的循环打印次数。在 *i* 的值从 0 到 4 的这段时间内，两个线程是交替运行的，但当计数器 *i* 的取值变为 5 的时候，程序调用了 *TestThread* 类的 *stopMe* 方法，而在 *stopMe* 方法中，将 *bFlag* 变量赋值为 *false*，也就是终止了 *while* 循环，*run* 方法结束，*Thread-0* 线程随之结束。*main* 线程在计数器 *i* 等于 5 的时候，调用了 *TestThread* 类的 *stopMe* 方法后，CPU 不一定会马上切换到 *Thread-0* 线程上，也就是说 *Thread-0* 线程不一定会马上终止，*main* 线程的计数器 *i* 可能还会继续累加，之后 *Thread-0* 线程才真正结束。

综上所述，通过控制 *run* 方法中循环条件的方式来结束一个线程的方法是值得推荐使用的办法，这也是实际中用的最多的方法。

19.8 练一练

一、填空题

1. 多线程的优点为_____。
2. 返回线程的优先级的方法为_____。
3. 死锁的形成原因为_____。

二、简答题

1. 简述线程和进程的主要差别。
2. 简述激活线程的方法。

19.9 跟我上机

编写一个多线程处理的程序，其他线程运行 10 秒后，使用 main 方法中断其他线程。

第 20 章

文件IO操作



本章视频教学录像：4 小时 40 分钟

要把程序数据保存到文件中，就一定要使用I/O输入输出技术。Java中提供的I/O操作可以把数据保存到多种类型的文件中。本章讲解文件I/O操作的File类、各种流类、字符的编码以及对象序列化的相关知识。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握文件 I/O 操作的相关概念
- ☐ 熟悉 Java 中的各种流类
- ☐ 了解字符的编码
- ☐ 了解对象的序列化



大多数的应用程序都需要与外部设备进行数据交换，最常见的外部设备包含磁盘和网络。IO 就是指应用程序对这些设备的数据输入与输出。在程序中，键盘被用做文件输入，显示器被用做文件输出。JAVA 语言定义了许多类专门负责各种方式的输入输出，这些类都被放在 java.io 包中。

20.1 File 类

▶ 本节视频教学录像：39 分钟

File 类是 IO 包中唯一代表磁盘文件本身的对象。File 类定义了一些与平台无关的方法来操纵文件，通过调用 File 类提供的各种方法，能够完成创建、删除文件，重命名文件，判断文件的读写权限及文件是否存在，设置和查询文件的最近修改时间等操作。

Java 能正确处理 UNIX 和 Windows/DOS 约定路径分隔符。如果在 Windows 版本的 Java 下用斜线 (/)，路径处理依然正确。记住：如果 Windows/DOS 使用反斜线 (\) 的约定，就需要在字符串内使用它的转义序列 (\\)。Java 约定是用 UNIX 和 URL 风格的斜线来作路径分隔符。

下面的构造方法可以用来生成 File 对象。

File(String directoryPath)

在这里，“directoryPath”是文件的路径名。

File 定义了很多获取 File 对象标准属性的方法。例如 getName() 用于返回文件名，getParent() 返回父目录名；exists() 方法在文件存在的情况下返回 true，反之返回 false。然而 File 类是不对称的，意思是虽然存在可以验证一个简单文件对象属性的很多方法，但是没有相应的方法来改变这些属性。下面的例子说明了 File 的几个方法。

【范例 20-1】 File 方法的使用（代码 20-1.txt）。

```
01 import java.io.*;
02 public class FileDemo
03 {
04     public static void main(String[] args)
05     {
06         File f=new File("c:\\1.txt");
07         if(f.exists())
08             f.delete();
09         else
10             try
11             {
12                 f.createNewFile();
13             }
14             catch(Exception e)
15             {
16                 System.out.println(e.getMessage());
17             }
18     }
19 }
```

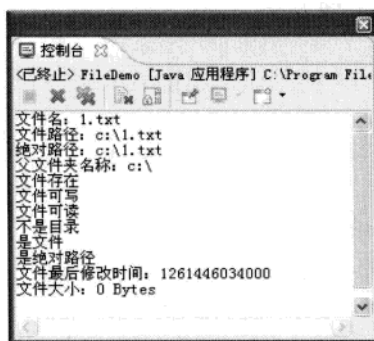
```

18      // getName()方法, 取得文件名
19      System.out.println("文件名: "+f.getName());
20      // getPath()方法, 取得文件路径
21      System.out.println("文件路径: "+f.getPath());
22      // getAbsolutePath()方法, 得到绝对路径名
23      System.out.println("绝对路径: "+f.getAbsolutePath());
24      // getParent()方法, 得到父文件夹名
25      System.out.println("父文件夹名称: "+f.getParent());
26      // exists(), 判断文件是否存在
27      System.out.println(f.exists()?"文件存在":"文件不存在");
28      // canWrite(), 判断文件是否可写
29      System.out.println(f.canWrite()?"文件可写":"文件不可写");
30      // canRead(), 判断文件是否可读
31      System.out.println(f.canRead()?"文件可读":"文件不可读");
32      // isDirectory(), 判断是否是目录
33      System.out.println(f.isDirectory()?"是":"不是"+"目录");
34      // isFile(), 判断是否是文件
35      System.out.println(f.isFile()?"是文件":"不是文件");
36      // isAbsolute(), 是否是绝对路径名称
37      System.out.println(f.isAbsolute()?"是绝对路径":"不是绝对路径");
38      // lastModified(), 文件最后的修改时间
39      System.out.println("文件最后修改时间: "+f.lastModified());
40      // length(), 文件的长度
41      System.out.println("文件大小: "+f.length()+" Bytes");
42  }
43  }

```

【运行结果】

保存并运行程序, 结果如图所示。



在 File 类中还有许多的方法, 读者没有必要去死记这些用法, 只要记住在需要的时候去查 Java 的 API 手册就可以了。

20.2 RandomAccessFile 类

本节视频教学录像：18 分钟

RandomAccessFile 类可以说是 Java 语言中功能最为丰富的文件访问类，它提供了众多的文件访问方法。RandomAccessFile 类支持“随机访问”方式，可以跳转到文件的任意位置处读写数据。在要访问一个文件的时候，不想把文件从头读到尾，而是希望像访问一个数据库一样访问一个文本文件，这时使用 RandomAccessFile 类就是最佳选择。

RandomAccessFile 对象类有个位置指示器，指向当前读写处的位置，当读写 n 个字节后，文件指示器将指向这 n 个字节后面的下一个字节处。刚打开文件时，文件指示器指向文件的开头处，可以移动文件指示器到新的位置，随后的读写操作将从新的位置开始。RandomAccessFile 在数据等长记录格式文件的随机（相对顺序而言）读取时有很大的优势，但该类仅限于操作文件，不能访问其他的 IO 设备，如网络、内存映像等。

有关 RandomAccessFile 类中的成员方法及使用说明请参阅 JDK 文档。下面是一个使用 RandomAccessFile 的例子，往文件中写入 3 名员工的信息，然后按照第 2 名员工、第 1 名员工、第 3 名员工的先后顺序读出。RandomAccessFile 可以以只读或读写方式打开文件，具体使用哪种方式取决于用户创建 RandomAccessFile 类对象的构造方法。

```
new RandomAccessFile(f,"rw");           // 读写方式
new RandomAccessFile(f,"r");             // 只读方式
```



注意：当程序需要以读写的方式打开一个文件时，如果这个文件不存在，程序会自动创建此文件。

这里还需要设计一个类来封装员工信息。一个员工信息就是文件中的一条记录，而且必须保证每条记录在文件中的大小相同，也就是每个员工的姓名字段在文件中的长度是一样的，这样才能够准确定位每条记录在文件中的具体位置。假设 name 中有 8 个字符，少于 8 个则补空格(这里用“\u0000”)，多于 8 个则去掉后面多余的部分。由于年龄是整型数，所以不管这个数有多大，只要它不超过整型数的范围，在内存中都是占 4 个字节大小。

【范例 20-2】 员工信息类的使用（代码 20-2.txt）。

```
01  import java.io.*;
02  public class RandomFileDemo
03  {
04      public static void main(String [] args) throws Exception
05      {
06          Employee e1 = new Employee("zhangsan",23);
07          Employee e2 = new Employee("lisi",24);
08          Employee e3 = new Employee("wangwu",25);
09          RandomAccessFile ra=new RandomAccessFile("c:\\employee.txt","rw");
10          ra.write(e1.name.getBytes());
```

```

11     ra.writeInt(e1.age);
12     ra.write(e2.name.getBytes());
13     ra.writeInt(e2.age);
14     ra.write(e3.name.getBytes());
15     ra.writeInt(e3.age);
16     ra.close();
17     RandomAccessFile raf=new RandomAccessFile("c:\\employee.txt","r");
18     int len=8;
19     raf.skipBytes(12);           // 跳过第 1 个员工的信息，其姓名 8 字节，年龄 4 字节
20     System.out.println("第 2 个员工信息:");
21     String str="";
22     for(int i=0;i<len;i++)
23         str=str+(char)raf.readByte();
24     System.out.println("name:"+str);
25     System.out.println("age:"+raf.readInt());
26     System.out.println("第 1 个员工的信息:");
27     raf.seek(0);                // 将文件指针移动到文件开始位置
28     str="";
29     for(int i=0;i<len;i++)
30         str=str+(char)raf.readByte();
31     System.out.println("name:"+str);
32     System.out.println("age:"+raf.readInt());
33     System.out.println("第 3 个员工的信息:");
34     raf.skipBytes(12);          // 跳过第 2 个员工的信息
35     str="";
36     for(int i=0;i<len;i++)
37         str=str+(char)raf.readByte();
38     System.out.println("name:"+str.trim());
39     System.out.println("age:"+raf.readInt());
40     raf.close();
41 }
42 }
43 class Employee
44 {
45     String name;
46     int age;
47     final static int LEN=8;
48     public Employee(String name,int age)
49     {
50         if(name.length()>LEN)
51         {

```



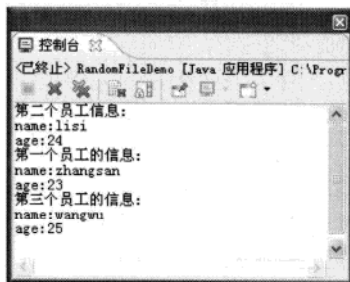
```

52         name=name.substring(0,8);
53     }
54     else
55     {
56         while(name.length()<LEN)
57             name=name+"\u0000";
58     }
59     this.name=name;
60     this.age=age;
61 }
62 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

本程序完成了所要实现的功能，显示出了 RandomAccessFile 类的作用。String.substring(int beginIndex,int endIndex)方法可以用于取出一个字符串中的部分子字符串，但要注意的一个细节是：子字符串中的第 1 个字符对应的是原字符串中的脚标为 beginIndex 处的字符，但最后的字符对应的是原字符串中的脚标为 endIndex-1 处的字符，而不是 endIndex 处的字符。

20.3 流类

▶ 本节视频教学录像：2 小时 26 分钟

Java 的流式输入/输出建立在 4 个抽象类的基础上：InputStream、OutputStream、Reader 和 Writer。它们用来创建具体流式子类。尽管程序通过具体子类进行输入/输出操作，但顶层的类定义了所有流类的基本通用功能。

InputStream 和 OutputStream 被设计成字节流类，而 Reader 和 Writer 被设计成字符流类。字节流类和字符流类形成分离的层次结构。一般来说，处理字符或字符串时应使用字符流类，处理字节或二进制对象时应使用字节流类。

一般在操作文件流时，不管是字节流还是字符流，都可以按照以下的方式进行。

- (1) 使用 File 类找到一个文件。
- (2) 通过 File 类的对象去实例化字节流或字符流的子类。

- (3) 进行字节（字符）的读、写操作。
- (4) 关闭文件流。

20.3.1 字节流

字节流类为处理字节式输入/输出提供了丰富的环境。一个字节流可以和其他任何类型的对象并用，包括二进制数据。这样的多功能性使得字节流对很多类型的程序都很重要。因为字节流类以 `InputStream` 和 `OutputStream` 为顶层，所以下面就从讨论这两个类开始。

1. `InputStream`（输入字节流）

`InputStream` 是一个定义了 Java 流式字节输入模式的抽象类，该类的所有方法在出错时都会引发一个 `IOException` 异常。表中显示了 `InputStream` 的方法。

方法	描述
<code>int available()</code>	返回当前可读的输入字节数
<code>void close()</code>	关闭输入流。关闭之后若再读取则会产生 <code>IOException</code> 异常
<code>void mark(int numBytes)</code>	在输入流的当前点放置一个标记。该流在读取 N 个 Bytes 字节前都保持有效
<code>boolean markSupported()</code>	如果调用的流支持 <code>mark()</code> / <code>reset()</code> 就返回 <code>true</code>
<code>int read()</code>	如果下一个字节可读则返回一个整型，遇到文件尾时返回 -1
<code>int read(byte buffer[])</code>	试图读取 <code>buffer.length</code> 个字节到 <code>buffer</code> 中，并返回实际成功读取的字节数。遇到文件尾时返回 -1
<code>int read(byte buffer[], int offset, int numBytes)</code>	试图读取 <code>buffer</code> 中从 <code>buffer[offset]</code> 开始的 <code>numBytes</code> 个字节，返回实际读取的字节数。遇到文件结束时返回 -1
<code>void reset()</code>	重新设置输入指针到先前设置的标志处
<code>long skip(long numBytes)</code>	忽略 <code>numBytes</code> 个输入字节，返回实际忽略的字节数

2. `OutputStream`（输出字节流）

`OutputStream` 是定义了流式字节输出模式的抽象类，该类的所有方法返回一个 `void` 值并且在出错的情况下引发一个 `IOException` 异常。表中显示了 `OutputStream` 的方法。

方法	描述
<code>void close()</code>	关闭输出流。关闭后的写操作会产生 <code>IOException</code> 异常
<code>void flush()</code>	定制输出状态以使每个缓冲器都被清除，也就是刷新输出缓冲区
<code>void write(int b)</code>	向输出流写入单个字节。注意参数是一个整型数，它允许设计者不必把参数转换成字节型就可以调用 <code>write()</code> 方法
<code>void write(byte buffer[])</code>	向一个输出流写一个完整的字节数组
<code>void write(byte buffer[], int offset, int numBytes)</code>	写数组 <code>buffer</code> 以 <code>buffer[offset]</code> 为起点的 <code>numBytes</code> 个字节区域内的内容



注意：上两个表中的多数方法由 `InputStream` 和 `OutputStream` 的子类来实现，但 `mark()` 和 `reset()` 方法除外。注意下面讨论的每个子类中这些方法的使用和不使用的情况。

3. `FileInputStream`（文件输入流）

`FileInputStream` 类创建一个能从文件读取字节的 `InputStream` 类，它的两个常用的构造方法如下。

```
FileInputStream(String filepath)
```

```
FileInputStream(File fileObj)
```

这两个构造方法都能引发 `FileNotFoundException` 异常。在这里 `filepath` 是文件的绝对路径, `fileObj` 是描述该文件的 `File` 对象。

下面的例子创建了两个使用同样磁盘文件且各含一个上面所描述的构造方法的 `FileInputStream` 类。

```
InputStream f0 = new FileInputStream("c:\\test.txt");
File f = new File("c:\\test.txt");
InputStream f1 = new FileInputStream(f);
```

尽管第 1 个构造方法可能更常用到, 而第 2 个构造方法则允许在把文件赋给输入流之前用 `File` 方法更进一步检查文件。当一个 `FileInputStream` 被创建时, 它可以被公开读取。

4. `FileOutputStream` (文件输出流)

`FileOutputStream` 创建了一个可以向文件写入字节的类 `OutputStream`, 它常用的构造方法如下。

```
FileOutputStream(String filePath)
```

```
FileOutputStream(File fileObj)
```

```
FileOutputStream(String filePath, boolean append)
```

它们可以引发 `IOException` 或 `SecurityException` 异常。在这里 `filePath` 是文件的绝对路径, `fileObj` 是描述该文件的 `File` 对象。如果 `append` 为 `true`, 文件则以设置搜索路径模式打开。`FileOutputStream` 的创建不依赖于文件是否存在。在创建对象时, `FileOutputStream` 会在打开输出文件之前就创建它。在这种情况下如果试图打开一个只读文件, 则会引发一个 `IOException` 异常。

在下面的例子中, 用 `FileOutputStream` 类向文件中写入一个字符串, 并用 `FileInputStream` 读出写入的内容。

【范例 20-3】 向文件中写入字符串并读出 (代码 20-3.txt)。

```
01 import java.io.*;
02 public class StreamDemo
03 {
04     public static void main(String args[])
05     {
06         File f = new File("c:\\temp.txt");
07         OutputStream out = null;
08         try
09         {
10             out = new FileOutputStream(f);
11         }
12         catch (FileNotFoundException e)
```

```

13     {
14         e.printStackTrace();
15     }
16     // 将字符串转成字节数组
17     byte b[] = "Hello World!!!".getBytes();
18     try
19     {
20         // 将 byte 数组写入到文件之中
21         out.write(b);
22     }
23     catch (IOException e1)
24     {
25         e1.printStackTrace();
26     }
27     try
28     {
29         out.close();
30     }
31     catch (IOException e2)
32     {
33         e2.printStackTrace();
34     }
35
36     // 以下为读文件操作
37     InputStream in = null;
38     try
39     {
40         in = new FileInputStream(f);
41     }
42     catch (FileNotFoundException e3)
43     {
44         e3.printStackTrace();
45     }
46     // 开辟一个空间用于接收文件读进来的数据
47     byte b1[] = new byte[1024];
48     int i = 0;
49     try
50     {
51         // 将 b1 的引用传递到 read()方法之中, 同时此方法返回读入数据的个数
52         i = in.read(b1);
53     }

```

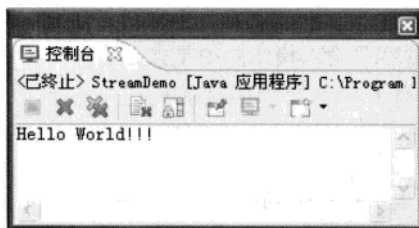
```

54         catch (IOException e4)
55         {
56             e4.printStackTrace();
57         }
58         try
59         {
60             in.close();
61         }
62         catch (IOException e5)
63         {
64             e5.printStackTrace();
65         }
66         //将 byte 数组转换为字符串输出
67         System.out.println(new String(b1,0,i));
68     }
69 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

此程序分为两个部分，一部分是向文件中写入内容（第 8~34 行），另一部分是从文件中读取内容（第 36~65 行）。

- (1) 第 6 行通过一个 File 类找到 C 盘下的一个 temp.txt 文件。
- (2) 向文件写入内容。

① 第 8~15 行通过 File 类的对象去实例化 OutputStream 的对象，此时是通过其子类 FileOutputStream 实例化的 OutputStream 对象，属于对象的向上转型。

② 因为字节流主要以操作 byte 数组为主，所以第 17 行通过 String 类中的 getBytes() 方法，将字符串转换成一个 byte 数组。

③ 第 18~26 行调用 OutputStream 类中的 write() 方法，将 byte 数组中的内容写入到文件中。

④ 第 27~34 行调用 OutputStream 类中的 close() 方法，关闭数据流操作。

- (3) 从文件中读入内容。

① 第 37~45 行通过 File 类的对象去实例化 InputStream 的对象，此时是通过其子类

FileInputStream 实例化的 InputStream 对象，属于对象的向上转型。

- ② 因为字节流主要以操作 byte 数组为主，所以第 47 行声明了一个 1024 大小的 byte 数组，此数组用于存放读入的数据。
- ③ 第 49~57 行调用 InputStream 类中的 read() 方法将文件中的内容读入到 byte 数组中，同时返回读入数据的个数。
- ④ 第 58~65 行调用 InputStream 类中的 close() 方法，关闭数据流操作。
- ⑤ 第 67 行将 byte 数组转成字符串输出。

【范例分析】

从本范例中可以看到，大部分的方法操作时都进行了异常处理，这是因为所使用的方法处都用 throws 关键字抛出了异常，所以在这里需要进行异常捕捉。不清楚的读者可以查找 JDK 文档，届时相信就可以明白了。

20.3.2 字符流

尽管字节流提供了处理任何类型输入/输出操作的足够的功能，但它们不能直接操作 Unicode 字符。既然 Java 的一个主要目标是支持“一次编写，处处运行”，那么包含直接的字符输入/输出的支持就是必要的。本小节讨论几个字符输入/输出类。如前所述，字符流层次结构的顶层是 Reader 和 Writer 抽象类，因此将从它们开始介绍。

1. Reader

Reader 是定义 Java 的流式字符输入模式的抽象类，该类的所有方法在出错的情况下都将引发 IOException 异常。表中给出了 Reader 类中的方法。

方法	描述
abstract void close()	关闭输入源。进一步的读取将会产生 IOException 异常
void mark(int numChars)	在输入流的当前位置设立一个标志。该输入流在 numChars 个字符被读取之前有效
boolean markSupported()	该流支持 mark()/reset() 则返回 true
int read()	如果调用的输入流的下一个字符可读则返回一个整型。遇到文件尾时返回 -1
int read(char buffer[])	试图读取 buffer 中的 buffer.length 个字符，返回实际成功读取的字符数。遇到文件尾返回 -1
abstract int read(char buffer[], int offset, int numChars)	试图读取 buffer 中从 buffer[offset] 开始的 numChars 个字符，返回实际成功读取的字符数。遇到文件尾返回 -1
boolean ready()	如果下一个输入请求不等待则返回 true，否则返回 false
long skip(long numChars)	跳过 numChars 个输入字符，返回跳过的字符设置输入指针到先前设立的标志处

2. Writer

Writer 是定义流式字符输出的抽象类，所有该类的方法都返回一个 void 值并在出错的条件引发 IOException 异常。表中给出了 Writer 类中方法。

方法	描述
abstract void close()	关闭输出流。关闭后的写操作会产生 IOException 异常
abstract void flush()	定制输出状态以使每个缓冲器都被清除。也就是刷新输出缓冲

续表

方法	描述
<code>void write(int ch)</code>	向输出流写入单个字符。注意参数是一个整型，它允许设计者不必把参数转换成字符型就可以调用 <code>write()</code> 方法
<code>void write(char buffer[])</code>	向一个输出流写一个完整的字符数组
<code>abstract void write(char buffer[],int offset,int numChars)</code>	向调用的输出流写入数组 <code>buffer</code> 以 <code>buffer[offset]</code> 为起点的 <code>N</code> 个 <code>Chars</code> 区域内的内容
<code>void write(String str)</code>	向调用的输出流写 <code>str</code>
<code>void write(String str, int offset,int numChars)</code>	写数组 <code>str</code> 中以指定的 <code>offset</code> 为起点的长度为 <code>numChars</code> 个字符区域内的内容

3. FileReader

`FileReader` 类创建了一个可以读取文件内容的 `Reader` 类。它最常用的构造方法如下。

```
FileReader(String filePath)
```

```
FileReader(File fileObj)
```

每一个都能引发一个 `FileNotFoundException` 异常。在这里 `filePath` 是一个文件的完整路径，`fileObj` 是描述该文件的 `File` 对象。

4. FileWriter

`FileWriter` 创建一个可以写文件的 `Writer` 类。它最常用的构造方法如下。

```
FileWriter(String filePath)
```

```
FileWriter(String filePath, boolean append)
```

```
FileWriter(File fileObj)
```

它们可以引发 `IOException` 或 `SecurityException` 异常。在这里 `filePath` 是文件的绝对路径，`fileObj` 是描述该文件的 `File` 对象。如果 `append` 为 `true`，输出是附加到文件尾的。`FileWriter` 类的创建不依赖于文件存在与否。在创建文件之前，`FileWriter` 将在创建对象时打开它来作为输出。如果试图打开一个只读文件，将引发一个 `IOException` 异常。

下面的例子是将上面的例题进行改写，先来看一下代码。

【范例 20-4】 字符流的使用（代码 20-4.txt）。

```

01  import java.io.*;
02  public class CharDemo
03  {
04      public static void main(String args[])
05      {
06          File f = new File("c:\\temp.txt");
07          Writer out = null;
08          try
09          {
10              out = new FileWriter(f);
11          }

```

```
12      catch (IOException e)
13      {
14          e.printStackTrace();
15      }
16      // 声明一个 String 类型对象
17      String str = "Hello World!!!";
18      try
19      {
20          // 将 str 内容写入到文件之中
21          out.write(str);
22      }
23      catch (IOException e1)
24      {
25          e1.printStackTrace();
26      }
27      try
28      {
29          out.close();
30      }
31      catch (IOException e2)
32      {
33          e2.printStackTrace();
34      }
35
36      // 以下为读文件操作
37      Reader in = null;
38      try
39      {
40          in = new FileReader(f);
41      }
42      catch (FileNotFoundException e3)
43      {
44          e3.printStackTrace();
45      }
46      // 开辟一个空间用于接收文件读进来的数据
47      char c1[] = new char[1024];
48      int i = 0;
49      try
50      {
51          // 将 c1 的引用传递到 read() 方法之中, 同时此方法返回读入数据的个数
52          i = in.read(c1);
```

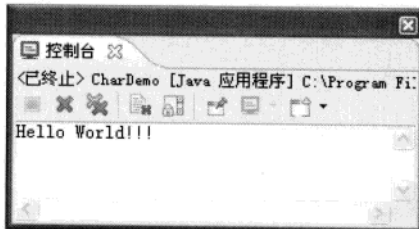
```

53     }
54     catch (IOException e4)
55     {
56         e4.printStackTrace();
57     }
58     try
59     {
60         in.close();
61     }
62     catch (IOException e5)
63     {
64         e5.printStackTrace();
65     }
66     //将字符数组转换为字符串输出
67     System.out.println(new String(c1,0,i));
68 }
69 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

此程序与上面范例的程序类似，也同样分为两部分，一部分是向文件中写入内容（第 8~34 行），另一部分是从文件中读取内容（第 36~65 行）。

(1) 第 6 行通过一个 File 类找到 C 盘下的一个 temp.txt 文件。

(2) 向文件写入内容。

① 第 7~15 行通过 File 类的对象去实例化 Writer 的对象，此时是通过其子类 FileWriter 实例化的 Writer 对象，属于对象的向上转型。

② 因为字符流主要以操作字符为主，所以第 17 行声明了一个 String 类的对象 str。

③ 第 18~26 行调用 Writer 类中的 write() 方法将字符串中的内容写入到文件中。

④ 第 27~34 行调用 Writer 类中的 close() 方法，关闭数据流操作。

(3) 从文件中读入内容。

① 第 37~45 行通过 File 类的对象去实例化 Reader 的对象，此时是通过其子类 FileReader 实例化的 Reader 对象，属于对象的向上转型。

- ② 因为字节流主要以操作 char 数组为主，所以第 47 行声明了一个 1024 大小的 char 数组，此数组用于存放读入的数据。
- ③ 第 49~57 行调用 Reader 类中的 read() 方法将文件中的内容读入到 char 数组中，同时返回读入数据的个数。
- ④ 第 58~65 行调用 Reader 类中的 close() 方法，关闭数据流操作。
- ⑤ 第 67 行将 char 数组转成字符串输出。



注 意：读者可以将范例 CharDemo 中的第 27~34 行注释掉，也就是说在向文件写入内容之后不关闭文件，然后打开文件，可以发现文件中没有任何内容，这是为什么？从 JDK 文档之中查找 FileWriter 类，如图所示。

```

java.io
Class FileWriter

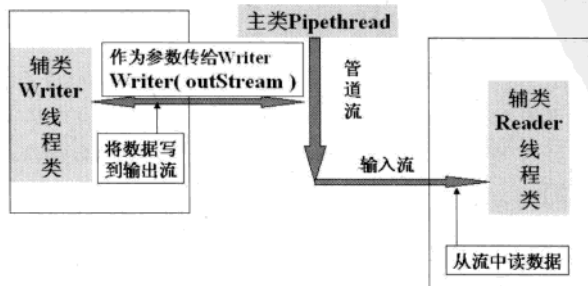
java.lang.Object
├─ java.io.Writer
│   └─ java.io.OutputStreamWriter
│       └─ java.io.FileWriter
    
```

可以看到，FileWriter 类并不是直接继承自 Writer 类，而是继承了 Writer 的子类（OutputStreamWriter），此类为字节流和字符流的转换类，后面会介绍。也就是说真正从文件中读取进来的数据还是字节，只是在内存中将字节转换成了字符。所以可得出一个结论：字符流用到了缓冲区，而字节流没有用到缓冲区。另外也可以用 Writer 类中的 flush() 方法强制清空缓冲区。

20.3.3 管道流

管道流主要用于连接两个线程间的通信。管道流也分为字节流（PipedInputStream、PipedOutputStream）与字符流（PipedReader、PipedWriter）两种类型，本小节主要讲解 PipedInputStream 和 PipedOutputStream。

一个 PipedInputStream 对象必须和一个 PipedOutputStream 对象进行连接而产生一个通信管道，PipedOutputStream 可以向管道中写入数据，PipedInputStream 可以从管道中读取 PipedOutputStream 写入的数据。如图所示，这两个类主要用来完成线程之间的通信，一个线程的 PipedInputStream 对象能够从另外一个线程的 PipedOutputStream 对象中读取数据。



【范例 20-5】 管道流的使用（代码 20-5.txt）。

```
01  import java.io.*;
02  public class PipeStreamDemo
03  {
04      public static void main(String args[])
05      {
06          try
07          {
08              Sender sender = new Sender();           // 产生两个线程对象
09              Receiver receiver = new Receiver();
10              PipedOutputStream out = sender.getOutputStream(); // 写入
11              PipedInputStream in = receiver.getInputStream();    // 读出
12              out.connect(in);                                   // 将输出发送到输入
13              sender.start();                                   // 启动线程
14              receiver.start();
15          }
16          catch(IOException e)
17          {
18              System.out.println(e.getMessage());
19          }
20      }
21  }
22  class Sender extends Thread
23  {
24      private PipedOutputStream out=new PipedOutputStream();
25      public PipedOutputStream getOutputStream()
26      {
27          return out;
28      }
29      public void run()
30      {
31          String s=new String("Receiver, 你好!");
32          try
33          {
34              out.write(s.getBytes());                // 写入（发送）
35              out.close();
36          }
37          catch(IOException e)
38          {
39              System.out.println(e.getMessage());
```



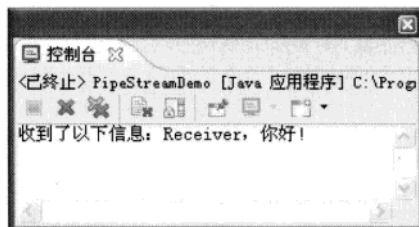
```

40     }
41 }
42 }
43 class Receiver extends Thread
44 {
45     private PipedInputStream in=new PipedInputStream();
46     public PipedInputStream getInputStream()
47     {
48         return in;
49     }
50     public void run()
51     {
52         String s=null;
53         byte [] buf = new byte[1024];
54         try
55         {
56             int len =in.read(buf);           // 读出数据
57             s = new String(buf,0,len);
58             System.out.println("收到了以下信息: "+s);
59             in.close();
60         }
61         catch(IOException e)
62         {
63             System.out.println(e.getMessage());
64         }
65     }
66 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 22~42 行声明了一个 Sender 类，此类继承自 Thread 类，所以此类覆写了 Runnable 接口

之中的 run() 方法。第 24 行声明了一个 PipedOutputStream 对象 out，此对象用于发送信息。

第 43~66 行声明了一个 Receiver 类，此类继承自 Thread 类，所以此类覆写了 Runnable 接口之中的 run() 方法。第 45 行声明了一个 PipedInputStream 对象 in，此对象用于接收其他线程发来的信息。

第 8 行和第 9 行分别声明了 Sender 和 Receiver 的实例化对象，之后返回各自的管道输出流及管道输入流对象，通过管道输出流的 connect 方法，将两个管道连接在一起，之后分别启动线程。

20.3.4 ByteArrayInputStream 与 ByteArrayOutputStream

ByteArrayInputStream 是输入流的一种实现，它有两个构造方法，每个构造方法都需要一个字节数组来作为其数据源。

```
ByteArrayInputStream(byte[] buf)
ByteArrayInputStream(byte[] buf, int offset, int length)
ByteArrayOutputStream()
ByteArrayOutputStream(int)
```

如果程序在运行的过程中要产生一些临时文件，可以采用虚拟文件方式实现，JDK 中提供有 ByteArrayInputStream 和 ByteArrayOutputStream 两个类可以实现类似于内存虚拟文件的功能。

【范例 20-6】 ByteArrayInputStream 类的使用（代码 20-6.txt）。

```
01 import java.io.*;
02 public class ByteArrayDemo {
03     public static void main(String[] args) throws Exception
04     {
05         String tmp = "abcdefghijklmnopqrstuvwxyz";
06         byte[] src = tmp.getBytes();           // src 为转换前的内存块
07         ByteArrayInputStream input = new ByteArrayInputStream(src);
08         ByteArrayOutputStream output = new ByteArrayOutputStream();
09         new ByteArrayDemo().transform(input, output);
10         byte[] result = output.toByteArray();  // result 为转换后的内存块
11         System.out.println(new String(result));
12     }
13     public void transform(InputStream in, OutputStream out)
14     {
15         int c = 0;
16         try
17         {
18             while ((c = in.read()) != -1)       // read 在读到流的结尾处返回-1
19             {
```

```
20         int C = (int) Character.toUpperCase((char) c);
21         out.write(C);
22     }
23 }
24 catch (Exception e)
25 {
26     e.printStackTrace();
27 }
28 }
29 }
```

20.3.5 System.in 和 System.out

为了支持标准输入输出设备，Java 定义了两个特殊的流对象：System.in 和 System.out。System.in 对应键盘，属于 InputStream 类型，程序使用 System.in 可以读取从键盘上输入的数据。System.out 对应显示器，属于 PrintStream 类型，PrintStream 是 OutputStream 的一个子类，程序使用 System.out 可以将数据输出到显示器上。键盘可以被当做一个特殊的输入流，显示器可以被当做一个特殊的输出流。

20.3.6 打印流

PrintStream 类提供了一系列的 print 和 println 方法，可以实现将基本数据类型的格式转换成字符串输出。在前面的程序中大量用到的“System.out.println”语句中的 System.out，就是 PrintStream 类的一个实例对象。PrintStream 有下面几个构造方法。

```
PrintStream(OutputStream out)
PrintStream(OutputStream out,boolean autoflush)
PrintStream(OutputStream out,boolean autoflush, String encoding)
```

其中 autoflush 控制在 Java 中遇到换行符(\n)时是否自动清空缓冲区，encoding 是指定编码方式。关于编码方式，将在本章后面介绍。

println 方法与 print 方法的区别是：前者会在打印完的内容后面再多打印一个换行符(\n)，所以 println() 等于 print("\n")。

Java 的 PrintStream 对象具有多个重载的 print 和 println 方法，它们可输出各种类型（包括 Object）的数据。对于基本数据类型的数据，print 和 println 方法会先将它们转换成字符串的形式，然后再输出，而不是输出原始的字节内容，如整数 221 的打印结果是字符“2”、“2”、“1”所组合成的一个字符串，而不是整数 221 在内存中的原始字节数据。对于一个非基本数据类型的对象，print 和 println 方法会先调用对象的 toString 方法，然后输出 toString 方法所返回的字符串。

IO 包中提供了一个与 PrintStream 对应的 PrintWriter 类，PrintWriter 类有下列几个构造方法。

```
PrintWriter(OutputStream)
```

```

PrintWriter(OutputStream, boolean)
PrintWriter(Writer)
PrintWriter(Writer, boolean)

```

PrintWriter 即使遇到换行符(\n)也不会自动清空缓冲区, 只在设置了 autoflush 模式下使用了 println 方法后才会自动清空缓冲区。PrintWriter 相对 PrintStream 最有利的一个地方就是 println 方法的行为, Windows 下的文本换行是 “\r\n”, 而 Linux 下的文本换行是 “\n”。如果希望程序能够生成平台相关的文本换行, 而不是在各种平台下都用 “\n” 作为文本换行, 那么就应该使用 PrintWriter 的 println 方法, PrintWriter 的 println 方法能根据不同的操作系统而生成相应的换行符。

下面的范例通过 PrintWriter 类向屏幕上打印信息。

【范例 20-7】 PrintWriter 类向屏幕输出信息 (代码 20-7.txt)。

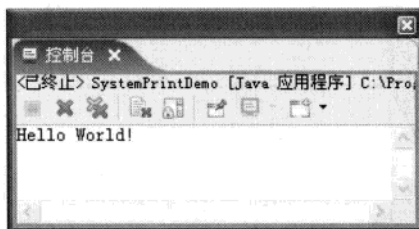
```

01  import java.io.*;
02  public class SystemPrintDemo
03  {
04      public static void main(String args[])
05      {
06          PrintWriter out = null;
07          // 通过 System.out 对 PrintWriter 实例化
08          out = new PrintWriter(System.out);
09          // 向屏幕上输出
10          out.print("Hello World!");
11          out.close();
12      }
13  }

```

【运行结果】

保存并运行程序, 结果如图所示。



【代码详解】

第 8 行通过 System.out 实例化 PrintWriter, 此时 PrintWriter 类的实例化对象 out 就具备了向屏幕输出信息的能力, 所以在第 10 行调用 print() 方法时, 就会将内容打印到屏幕上。

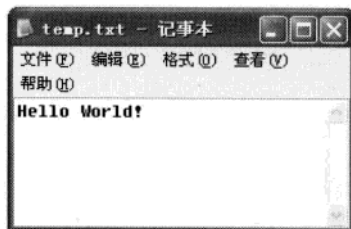
下面的范例通过 PrintWriter 向文件中打印信息。

【范例 20-8】 通过 PrintWriter 向文件中输出信息（代码 20-8.txt）。

```
01  import java.io.*;
02  public class FilePrint
03  {
04      public static void main(String args[])
05      {
06          PrintWriter out = null ;
07          File f = new File("c:\\temp.txt") ;
08          try
09          {
10              out = new PrintWriter(new FileWriter(f)) ;
11          }
12          catch (IOException e)
13          {
14              e.printStackTrace();
15          }
16          // 由 FileWriter 实例化，则向文件中输出
17          out. print ("Hello World!"+"\\r\\n");
18          out.close() ;
19      }
20  }
```

【运行结果】

保存并运行程序，结果如图所示。

**【代码详解】**

第 10 行通过 FileWriter 类实例化 PrintWriter，此时 PrintWriter 类的实例化对象 out 就具备了向文件输出信息的能力，所以在第 17 行调用 print()方法时，就会将内容输出到文件之中。

20.3.7 DataInputStream 与 DataOutputStream

DataInputStream 与 DataOutputStream 提供了与平台无关的数据操作，通常会先通过 DataOutputStream 按照一定的格式输出，再通过 DataInputStream 按照一定格式读入。由于可以

得到 java 的各种基本类型甚至字符串，这样对得到的数据便可以方便地进行处理，这在通过协议传输的信息的网络上是非常适用的。

如下面的范例。

用户的定单用如下格式储存为 order.txt 文件。

价格	数量	描述
18.99	10	T 恤衫
9.22	10	杯子

实现机制如下。

(1) 写入端构造一个 DataOutputStream，并按照一定的格式写入数据。

```
// 将数据写入某一种载体
DataOutputStream out = new DataOutputStream(new FileOutputStream("order.txt"));
// 价格
double[] prices = { 18.99, 9.22, 14.22, 5.22, 4.21 };
// 数目
int[] units = { 10, 10, 20, 39, 40 };
// 产品名称
String[] desc = { "T 恤衫", "杯子", "洋娃娃", "大头针", "钥匙链" };
// 向数据过滤流写入主要类型
for (int i = 0; i < prices.length; i++)
{
    // 写入价格，使用 tab 隔开数据
    out.writeDouble(prices[i]);
    out.writeChar('\t');
    // 写入数目
    out.writeInt(units[i]);
    out.writeChar('\t');
    // 写入产品名称，行尾加入换行符
    out.writeChars(desc[i]);
    out.writeChar('\n');
}
out.close();
```

(2) 计价程序读入并在标准输出中输出。

```
// 将数据读出
DataInputStream in = new DataInputStream(new FileInputStream("order.txt"));
double price;
int unit;
StringBuffer desc;
double total = 0.0;
try
```

```
{
    // 当文本被全部读出以后会抛出 EOFException 异常，中断循环
    while (true)
    {
        // 读出价格
        price = in.readDouble();
        // 跳过 tab
        in.readChar();
        // 读出数目
        unit = in.readInt();
        // 跳过 tab
        in.readChar();
        char chr;
        // 读出产品名称
        desc = new StringBuffer();
        while ((chr = in.readChar()) != '\n')
        {
            desc.append(chr);
        }
        System.out.println("订单信息: " + "产品名称: "+desc+" , \t 数量: "+unit+" , \t 价格: "+price);
        total = total + unit * price;
    }
}
catch (EOFException e)
{
    System.out.println("\n 总共需要: " + total+"元");
}
in.close();
```

【范例 20-9】 DataInputStream 与 DataOutputStream 的使用（代码 20-9.txt）。

```
01  import java.io.*;
02  public class DataStreamDemo
03  {
04      public static void main(String[] args) throws IOException
05      {
06          // 将数据写入某一种载体
07          DataOutputStream out = new DataOutputStream(new FileOutputStream("order.txt"));
08          // 价格
09          double[] prices = { 18.99, 9.22, 14.22, 5.22, 4.21 };
10          // 数目
```

```
11      int[] units = { 10, 10, 20, 39, 40 };
12      // 产品名称
13      String[] descs = { "T 恤衫", "杯子", "洋娃娃", "大头针", "钥匙链" };
14
15      // 向数据过滤流写入主要类型
16      for (int i = 0; i < prices.length; i++)
17      {
18          // 写入价格, 使用 tab 隔开数据
19          out.writeDouble(prices[i]);
20          out.writeChar('\t');
21          // 写入数目
22          out.writeInt(units[i]);
23          out.writeChar('\t');
24          // 写入产品名称, 行尾加入换行符
25          out.writeChars(descs[i]);
26          out.writeChar('\n');
27      }
28      out.close();
29
30      // 将数据读出
31      DataInputStream in = new DataInputStream(new FileInputStream("order.txt"));
32
33      double price;
34      int unit;
35      StringBuffer desc;
36      double total = 0.0;
37
38      try
39      {
40          // 当文本被全部读出以后会抛出 EOFException 异常, 中断循环
41          while (true)
42          {
43              // 读出价格
44              price = in.readDouble();
45              // 跳过 tab
46              in.readChar();
47              // 读出数目
48              unit = in.readInt();
49              // 跳过 tab
50              in.readChar();
51              char chr;
```

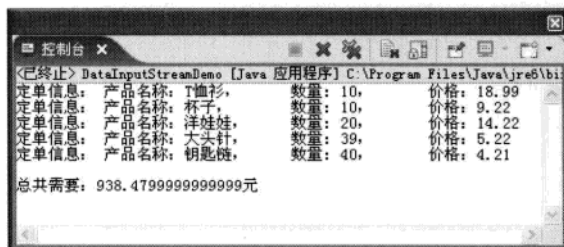
```

52         // 读出产品名称
53         desc = new StringBuffer();
54
55         while ((chr = in.readChar()) != '\n')
56         {
57             desc.append(chr);
58         }
59         System.out.println("订单信息: " + "产品名称: "+desc
60             +", \t 数量: +unit+", \t 价格: "+price);
61         total = total + unit * price;
62     }
63 }
64 catch (EOFException e)
65 {
66     System.out.println("\n 总共需要: " + total+"元");
67 }
68 in.close();
69 }
70 }

```

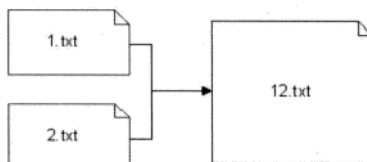
【运行结果】

保存并运行程序，结果如图所示。



20.3.8 合并流

采用 `SequenceInputStream` 类，可以实现两个文件的合并操作。如图所示。



【范例 20-10】 使用合并流将两个文件合并（代码 20-10.txt）。

```
01 import java.io.*;
```

382

```
02 public class SequenceDemo
03 {
04     public static void main(String[] args) throws IOException
05     {
06         // 声明两个文件读入流
07         FileInputStream in1 = null, in2 = null;
08         // 声明一个序列流
09         SequenceInputStream s = null;
10         FileOutputStream out = null;
11
12         try
13         {
14             // 构造两个被读入的文件
15             File inputFile1 = new File("c:\\1.txt");
16             File inputFile2 = new File("c:\\2.txt");
17             // 构造一个输出文件
18             File outputFile = new File("c:\\12.txt");
19
20             in1 = new FileInputStream(inputFile1);
21             in2 = new FileInputStream(inputFile2);
22
23             // 将两个输入流合为一个输入流
24             s = new SequenceInputStream(in1, in2);
25             out = new FileOutputStream(outputFile);
26
27             int c;
28             while ((c = s.read()) != -1)
29                 out.write(c);
30
31             in1.close();
32             in2.close();
33             s.close();
34             out.close();
35             System.out.println("ok...");
36         }
37         catch (IOException e)
38         {
39             e.printStackTrace();
40         }
41         finally
42         {
```



```
43         if (in1 != null)
44             try {
45                 in1.close();
46             } catch (IOException e) {
47             }
48         if (in2 != null)
49             try {
50                 in2.close();
51             } catch (IOException e) {
52             }
53         if (s != null)
54             try {
55                 s.close();
56             } catch (IOException e) {
57             }
58         if (out != null)
59             try {
60                 out.close();
61             } catch (IOException e) {
62             }
63     }
64 }
65 }
```

【运行结果】

保存并运行程序，结果如图所示。



20.3.9 字节流与字符流的转换

前面已经讲过，Java 支持字节流和字符流，但有时需要在字节流和字符流之间转换。

InputStreamReader 和 OutputStreamWriter，这两个类是字节流和字符流之间相互转换的类，

InputStreamReader 用于将一个字节流中的字节解码成字符，OutputStreamWriter 用于将写入的字符编码成字节后写入一个字节流。

InputStreamReader 有两个主要的构造方法：

```
InputStreamReader(InputStream in)
```

// 用默认字符集创建一个 InputStreamReader 对象

```
InputStreamReader(InputStream in,String CharsetName)
```

// 接收已指定字符集名的字符串，并用该字符集创建对象

OutputStreamWriter 也有对应的两个主要的构造方法：

```
OutputStreamWriter(OutputStream in)
```

// 用默认字符集创建一个 OutputStreamWriter 对象

```
OutputStreamWriter(OutputStream in,String CharsetName)
```

// 接收已指定字符集名的字符串，并用该字符集创建 OutputStreamWriter 对象

为了达到最高的效率，避免频繁地进行字符与字节间的相互转换，最好不要直接使用这两个类来进行读写，而应尽量使用 BufferedWriter 类包装 OutputStreamWriter 类，用 BufferedReader 类包装 InputStreamReader 类。

```
BufferedWriter out=new BufferedWriter(newOutputStreamWriter(System.out));
```

```
BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
```

接着从一个实际的应用中来了解 InputStreamReader 的作用，怎样用一种简单的方式一下子就读取到键盘上输入的一整行字符呢？只要用下面的两行程序代码就可以解决这个问题。

```
BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
```

```
String strLine = in.readLine();
```

可见，构建 BufferedReader 对象时，必须传递一个 Reader 类型的对象作为参数，而键盘对应的 System.in 是一个 InputStream 类型的对象，所以这里需要用到一个 InputStreamReader 的转换类，将 System.in 转换成字符流之后，放入到字符流缓冲区之中，之后从缓冲区中每次读入一行数据。

```
import java.io.*;
```

```
public class class_name
```

//类名

```
{
```

```
public static void main(String args[]) throws IOException
```

```
{
```

```
BufferedReader buf;
```

//声明 buf 为 BufferedReader 类的对象

```
String str;
```

//声明 str 为 String 类型的对象

```
.....
```

```
buf=new BufferedReader(new InputStreamReader(System.in));
```

```
str=buf.readLine();
```

//读入字符串至 buf

```
.....
```

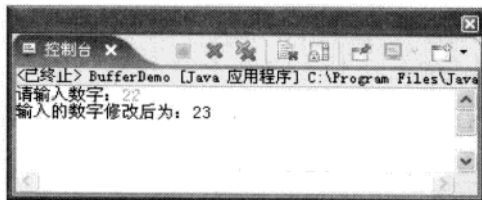
```
}  
}
```

【范例 20-11】 字符流的转换使用（代码 20-11.txt）。

```
01  import java.io.*;  
02  public class BufferDemo  
03  {  
04      public static void main(String args[])  
05      {  
06          BufferedReader buf = null;  
07          buf = new BufferedReader(new InputStreamReader(System.in));  
08          String str = null;  
09          while (true)  
10          {  
11              System.out.print("请输入数字：");  
12              try  
13              {  
14                  str = buf.readLine();  
15              } catch (IOException e)  
16              {  
17                  e.printStackTrace();  
18              }  
19              int i = -1;  
20              try  
21              {  
22                  i = Integer.parseInt(str);  
23                  i++;  
24                  System.out.println("输入的数字修改后为：" + i);  
25                  break;  
26              }  
27              catch (Exception e)  
28              {  
29                  System.out.println("输入的内容不正确，请重新输入！");  
30              }  
31          }  
32      }  
33  }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 6 行和第 7 行对 `BufferedReader` 对象实例化。因为现在需要从键盘输入数据，因此需要使用 `System.in` 进行实例化，但 `System.in` 是属于 `InputStream` 类型，所以使用 `InputStreamReader` 类将字节流转换成字符流，之后将字符流放入到 `BufferedReader` 中。

第 14 行通过 `BufferedReader` 类中的 `readLine()` 方法，等待键盘的输入数据。

第 22 行通过 `Integer` 类将输入的字符串转换成基本数据类型中的整型。

第 23 行将输入的数字加一。

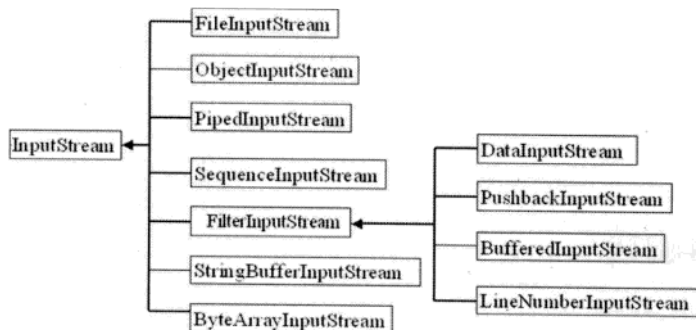
第 24 行输出修改后的数据。

20.3.10 IO 包中的类层次关系图

下面列出 IO 包中的各类层次关系图。

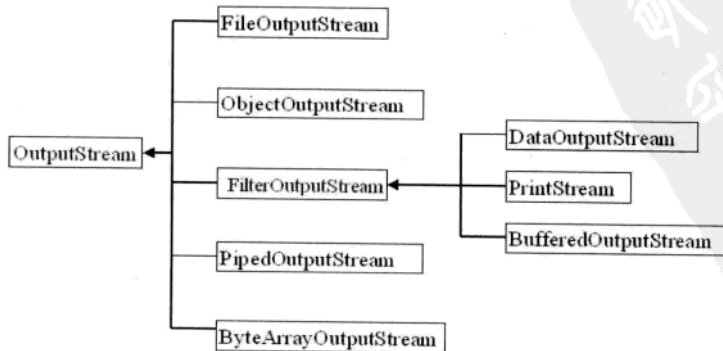
1. 字节输入流 (InputStream)

`InputStream` 类的层次结构如图所示。



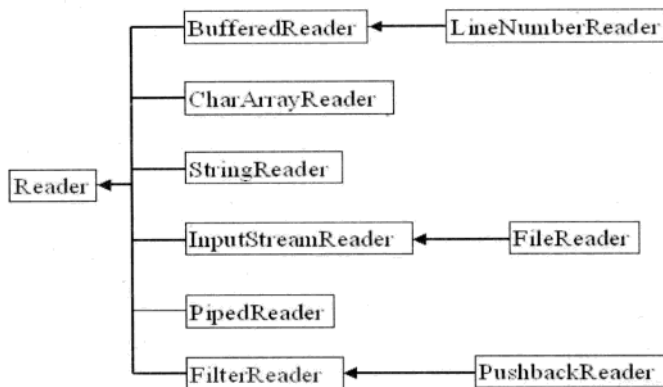
2. 字节输出流

`OutputStream` 类的层次结构如图所示。



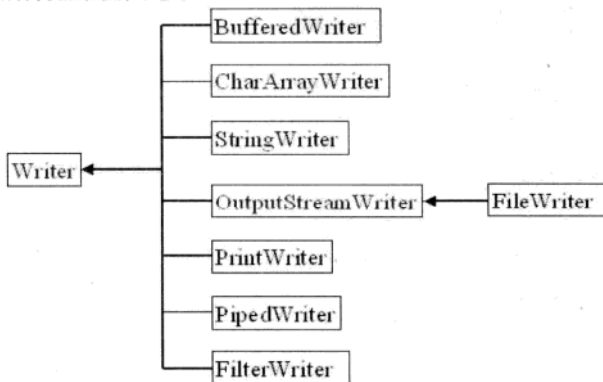
3. 字符输入流

Reader 类的层次结构如图所示。



4. 字符输出流

Writer 类的层次结构如图所示。



20.4 字符编码

▶ 本节视频教学录像：49 分钟

计算机里只有数字，计算机软件里的一切都是用数字来表示，屏幕上显示的一个个字符也不例外。最开始计算机是在美国使用，当时所用到的字符也就是现在键盘上的一些符号和少数几个特殊的符号，每一个字符都用一个数字来表示，一个字节所能表示的数字范围内足以容纳所有的字符，实际上表示这些字符的数字的字节最高位（bit）都为 0，也就是说这些数字都在 0 到 127 之间，如字符 a 对应数字 97，字符 b 对应数字 98 等，这种字符与数字对应的编码固定下来后，这套编码规则被称为 ASCII 码（美国标准信息交换码）。

随着计算机在其他国家的逐渐应用和普及，许多国家都把本地的字符集引入了计算机，这大大地扩展了计算机中字符的范围。一个字节所能表示的数字范围是不能容纳所有的中文汉字的。中国大陆将每一个中文字符都用两个字节的数字来表示，原有的 ASCII 码字符的编码保持不变，仍用一个字节表示。为了将一个中文字符与两个 ASCII 码字符相区别，中文字符的每个字节的最高位（bit）都为 1，中国大陆为每一个中文字符都指定了一个对应的数字，并作为标准的编码

固定了下来,这套编码规则称为 GBK(国标码),后来又在 GBK 的基础上对更多的中文字符(包括繁体)进行了编码,新的编码系统就是 GB2312,而 GBK 则是 GB2312 的子集。使用中文的国家和地区很多,同样的一个字符,如“中国”的“中”字,在中国大陆的编码是十六进制的 D6D0,而在中国台湾地区的编码则是十六进制的 A4A4,台湾地区对中文字符集的编码规则称为 BIG5(大五码)。

在一个国家的本地化系统中出现的一个字符,通过电子邮件传送到另外一个国家的本地化系统中,看到的就不是那个原始字符了,而是另外那个国家的一个字符或乱码。这是因为计算机里面并没有真正的字符,字符都是以数字的形式存在的,通过邮件传送一个字符,实际上传送的是这个字符对应的编码数字,同一个数字在不同的国家和地区代表的很可能是不同的符号。如十六进制的 D6D0 在中国大陆的本地化系统中显示为“中”这个符号,但在伊拉克的本地化系统中就不知道对应的是一个什么样的伊拉克字符了,反正人们看到的不是“中”这个符号。随着世界各国的交往越来越密切,全球一体化的趋势越来越明显,人们不可能完全忘记母语,都去使用英文在不同的国家和地区间交换越来越多的电子文档。特别是人们开发的应用软件都希望能走出国门、走向世界,可见,各个国家和地区都使用各自不同的本地化字符编码,已经给生活和工作带来了很大的不方便,严重制约了国家和地区间在计算机使用和技术方面的交流。

为了解决各个国家和地区使用各自不同的本地化字符编码带来的不便,人们将全世界所有的符号进行了统一编码,称之为 Unicode 编码。所有的字符不再区分国家和地区,都是人类共有的符号,如“中国”的“中”这个符号,在全世界的任何一个角落始终对应的都是一个十六进制的数字 4e2d。如果所有的计算机系统都使用这种编码方式,在中国大陆的本地化系统中显示的“中”这个符号,发送到德国的本地化系统中,显示的仍然是“中”这个符号,至于那个德国人能不能认识这个符号,就不是计算机所要解决的问题了。Unicode 编码的字符都占用两个字节的大小,也就是说全世界所有的字符个数不会超过 2 的 16 次方(65536),据推测一定是 Unicode 编码中没有包括诸如中国的藏文和满文这些少数民族的文字。

长期养成的保守习惯不可能一下子就改变过来,特别是不可能完全推翻那些已经存在的运行良好的系统。新开发的软件要做到瞻前顾后,既能够在存在的系统上运行,又便于以后的战略扩张和适应新的形式。Unicode 一统天下的局面暂时还难以形成,在相当长的一段时期内,人们看到的都是本地化字符编码与 Unicode 编码共存的景象。既然本地化字符编码与 Unicode 编码共存,那就少不了涉及两者之间的转换问题,而 Java 中的字符使用的都是 Unicode 编码,Java 技术在通过 Unicode 保证跨平台特性的前提下也支持了全扩展的本地平台字符集,而显示输出和键盘输入则都是采用的本地编码。

通过下面的程序,来看一下到底什么是字符乱码问题。在这里使用 String 类中的 getBytes()方法,对字符进行编码转换。

【范例 20-12】 字符编码使用范例 1 (代码 20-12.txt)。

```
01 import java.io.*;
02 public class EncodingDemo
03 {
04     public static void main(String args[]) throws Exception
05     {
06         // 在这里将字符串通过 getBytes()方法, 编码成 GB2312
```

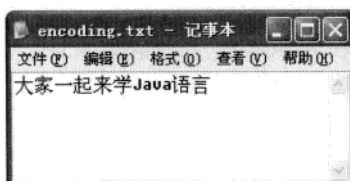
```

07     byte b[] = "大家一起学 Java 语言".getBytes("GB2312");
08     OutputStream out = new FileOutputStream(new File("c:\\encoding.txt"));
09     out.write(b);
10     out.close();
11 }
12 }

```

【运行结果】

保存并运行程序，结果如图所示。



对此程序读者应该非常清楚，但这里与之前稍有不同的是，在将字符串转换成 byte 数组的时候，用到了“GB2312”编码。

读到这里读者可能还是无法体会到字符编码问题，那么现在修改一下 EncodingDemo 程序，将字符编码转换成 ISO8859-1，但在执行此程序之前，须先执行下面的程序。

【范例 20-13】 字符编码使用范例 2（代码 20-13.txt）。

```

01  public class SetDemo
02  {
03      public static void main(String args[])
04      {
05          System.getProperties().put("file.encoding", "GB2312");
06      }
07  }

```

执行此程序之后，再运行 EncodingDemo.java 程序，修改后的程序如下。

【范例 20-14】 字符编码使用范例 3（代码 20-14.txt）。

```

01  import java.io.*;
02  public class EncodingDemo
03  {
04      public static void main(String args[]) throws Exception
05      {
06          // 在这里将字符串通过 getBytes()方法，编码成 ISO8859-1
07          byte b[] = "大家一起学 Java 语言".getBytes("ISO8859-1");
08          OutputStream out = new FileOutputStream(new File("c:\\encoding.txt"));
09          out.write(b);

```

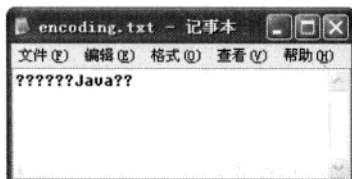
```

10      out.close();
11    }
12  }

```

【运行结果】

保存并运行程序，结果如图所示。



可以看到输出结果出现了乱码，这是为什么？这就是本节要讨论的字符编码问题。之所以会产生这样的问题，是因为在运行这段代码之前，先运行了 setDemo.java 程序，此程序主要是用来设置 JDK 环境的编码问题，所以乱码问题主要是由于 JDK 设置环境所引起的，为什么呢？读者可以运行下面的程序，观察其输出就可以发现问题。

【范例 20-15】 获得系统的属性（代码 20-15.txt）。

```

01  public class GetDemo
02  {
03      public static void main(String args[])
04      {
05          // 输出全部环境变量
06          System.getProperties().list(System.out);
07      }
08  }

```

【运行结果】

保存并运行程序，结果如图所示。



可以看到，在环境变量之中有一个 `file.encoding=GBK`，这清楚地表明了所使用的是 GBK 编码，而修改过的 `EncodingDemo.java` 程序中的第 7 行如下。

```
byte b[] = "大家一起来学 Java 语言".getBytes("ISO8859-1");
```

在这里将字符串编码换成了 ISO8859-1 编码。从前面介绍的基本知识已经知道 ISO8859-1 编码要大于 GBK 编码，所以才造成了字符的乱码问题。

20.5 对象序列化

▶ 本节视频教学录像：28 分钟

所谓的对象序列化（在某些书中也叫串行化），是指将对象转换成二进制数据流的一种实现手段。通过将对象序列化，可以方便地实现对象的传输及保存。

在 Java 中提供有 `ObjectInputStream` 与 `ObjectOutputStream` 这两个类用于序列化对象的操作。这两个类是用于存储和读取对象的输入输出流类，不难想象，只要把对象中的所有成员变量都存储起来，就等于保存了这个对象，之后从保存的对象之中再将对象读取进来就可以继续使用此对象。`ObjectInputStream` 类与 `ObjectOutputStream` 类，用于帮助开发者完成保存和读取对象成员变量取值的过程，但要求读写或存储的对象必须实现了 `Serializable` 接口，但 `Serializable` 接口中没有定义任何方法，仅仅被用做一种标记，以被编译器作特殊处理。如下范例所示。

【范例 20-16】 对象序列化使用范例 1（代码 20-16.txt）。

```
01  import java.io.*;
02  public class Person implements Serializable
03  {
04      private String name ;
05      private int age ;
06      public Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;
10      }
11      public String toString()
12      {
13          return " 姓名: "+this.name+" , 年龄: "+this.age ;
14      }
15  };
```

【代码详解】

在第 2 行中，类 `Person` 实现了 `Serializable` 接口，所以此类的对象可序列化。下面的范例使用 `ObjectOutputStream` 与 `ObjectInputStream` 将 `Person` 类的对象保存在文件之中。

【范例 20-17】 对象序列化使用范例 2（代码 20-17.txt）。

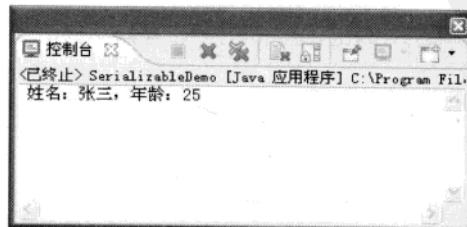
```

01  import java.io.*;
02  public class SerializableDemo
03  {
04      public static void main( String args[] ) throws Exception
05      {
06          File f = new File("SerializedPerson");
07          serialize(f);
08          deserialize(f);
09      }
10
11      // 以下方法为序列化对象方法
12      public static void serialize(File f) throws Exception
13      {
14          OutputStream outputFile = new FileOutputStream(f);
15          ObjectOutputStream cout = new ObjectOutputStream(outputFile);
16          cout.writeObject(new Person("张三",25));
17          cout.close();
18      }
19      // 以下方法为反序列化对象方法
20      public static void deserialize(File f) throws Exception
21      {
22          InputStream inputFile = new FileInputStream(f);
23          ObjectInputStream cin = new ObjectInputStream(inputFile);
24          Person p = (Person) cin.readObject();
25          System.out.println(p);
26      }
27  }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 12~18 行声明了一个 serialize() 方法，此方法用于将对象保存在文件之中。第 14 行、第

15 行为 `ObjectOutputStream` 对象实例化，此对象是通过 `FileOutputStream` 对象实例化，所以此类在保存 `Person` 对象时，向文件中输出。

第 20~26 行声明了一个 `deserialize()` 方法，此方法用于从文件中读取已经保存的对象。第 22 行、第 23 行为 `ObjectInputStream` 对象实例化。第 24 行调用 `ObjectInputStream` 类中的 `readObject()` 方法，从文件中读入内容，之后将读入的内容转型为 `Person` 类的实例。第 25 行直接打印 `Person` 对象实例，在打印对象时，默认调用 `Person` 类中的 `toString()` 方法。

另外要告诉读者的是，如果不希望类中的某个属性被序列化，可以在声明属性之前加上 `transient` 关键字。下面的代码修改自【范例 20-16】，在声明属性时，前面多加了一个 `transient` 关键字。

```
04     private transient String name ;
05     private transient int age ;
```

再次运行 `SerializableDemo.java` 程序时，其输出结果为：

姓名：null，年龄：0

从输出结果可以看到，`Person` 类中的两个属性并没有被保存下来，输出时，是直接输出了其默认值。

20.6 练一练

一、填空题

1. 在 Java 中要进行 IO 操作，需要导入_____包。
2. Java 中的数据操作主要分为_____和_____两种。
3. 生成 `File` 对象的构造方法为_____。

二、简答题

在操作文件流时，是按照怎样的方式进行的？

20.7 跟我上机

编写一个程序，向文件中写入“Here is my file.”，并从文件中读出字符串。

第 21 章

Java网页小程序——Java Applet



本章视频教学录像：7 分钟

Java Applet是编译过的Java程序，可以在所有支持Java的浏览器中运行。Applet程序可用于显示各种信息，还可以用来接受用户输入、处理输入。由于其跨平台、跨操作系统，所以得到了广泛的应用。本章讲解Applet程序的相关概念、使用Applet程序的基本方法、如何在HTML代码中嵌入Applet程序等内容。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握 Applet 程序的相关概念
- ☐ 掌握使用 Applet 程序的基本方法
- ☐ 掌握在 HTML 中嵌入 Applet 程序的方法
- ☐ 了解 Applet 标记



21.1 Applet 程序简介

▶ 本节视频教学录像：7 分钟

Applet 程序是一个经过编译的 Java 程序，它既可以在 Appletviewer 下运行，也可以在支持 Java 的 Web 浏览器中运行。Applet 程序可以完成图形显示、声音演奏、接受用户输入、处理输入内容等工作。Applet 程序中必须有一个是 Applet 类的子类。

Applet 程序能跨平台、跨操作系统、跨网络运行，因此它在 Internet 和 www 中得到了广泛的应用。另外，由于 Applet 程序代码小，易于快速地下载和发送，并且具有不需要修改应用程序就可以增加 Web 页新功能的特性，因此备受用户青睐。

Applet 程序不能单独运行，必须通过 HTML 调入后方能执行，以实现其功能。

下面先来看一个关于 Applet 的简单范例。

【范例 21-1】 Applet 的使用 (ch21\HelloApplet)。

```
01 package test ;
02 import java.awt.Graphics;
03 import java.applet.Applet;
04 public class HelloApplet extends Applet
05 {
06     public void paint(Graphics g)
07     {
08         g.drawString("Hello World!!",5,30);           //输出字符串
09     }
10 }
```

要运行 Applet 程序只有 java 程序是不够的，还需要另外编写 html 文件，将 Applet 程序嵌入到 html 文件之中。

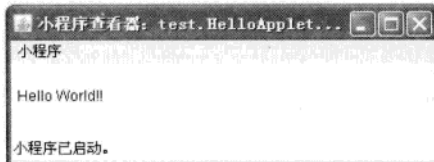
【范例 21-2】 Applet 程序 html 文件的编写 (ch21\HelloApplet)。

```
<HTML>
  <HEAD>
    <TITLE> Applet 程序 </TITLE>
  <BODY>
    <APPLET CODE="test.HelloApplet" WIDTH="300" HEIGHT="100">
    </APPLET>
  </BODY>
</HTML>
```

之后在命令行下执行 appletviewer hello.html 文件。

【运行结果】

保存并运行程序，结果如图所示。



21.2 Applet 程序中使用的几个基本方法

Applet 类是浏览器类库中最为重要的类，同时也是所有 Java 小应用程序的基本类。Applet 类中只有一种格式的构造方法 `public Applet ()`，此种方法用来创建一个 Applet 类的实例。因此，在编写 Applet 程序时，首先必须引入 `java.applet.Applet` 包。

一个 Applet 应用程序从开始运行到结束所经历的过程被称为 Applet 的生命周期。Applet 的生命周期涉及 `init()`、`start()`、`stop()` 和 `destroy()` 等 4 种方法，这 4 种方法都是 Applet 类的成员，可以继承这些方法，也可以重写这些方法，覆写原来定义的这些方法。除此之外，为了在 Applet 程序中实现输出功能，每个 Applet 程序中还需要重载 `paint()` 方法。

值得注意的是：在 Applet 类中没有提供 `init()`、`start()`、`stop()`、`destroy()` 和 `paint()` 等方法的任何实现，且它们都是被浏览器或 `appletviewer` 调用的，所以这几个方法要完成的功能应由编程人员自行编制。

1. `public void init()`

`init()` 方法是 Applet 运行的起点。当启动 Applet 程序时，系统首先调用此方法，以执行初始化任务。

2. `public void start()`

`start()` 方法是表明 Applet 程序开始执行的方法。当含有此 Applet 程序的 Web 页被再次访问时调用此方法。因此，如果每次访问 Web 页都需要进行一些操作的话，就需要在 Applet 程序中重载该方法。在 Applet 程序中，系统总是先调用 `init()` 方法，后调用 `start()` 方法。

3. `public void stop()`

`stop()` 方法用于使 Applet 停止执行。当含有该 Applet 的 Web 页被其他页代替时也要调用该方法。

4. `public void destroy()`

`destroy()` 方法收回 Applet 程序的所有资源，即释放已分配给它的所有资源。在 Applet 程序中，系统总是先调用 `stop()` 方法，后调用 `destroy()` 方法。

5. `paint(Graphics g)`

`paint(Graphics g)` 方法用于使 Applet 程序在屏幕上显示某些信息，如文字、色彩、背景或图像等。参数 `g` 是 `Graphics` 类的一个对象实例，实际上可以把 `g` 理解为一个画笔。对象 `g` 中包含了许多绘制方法，如 `drawString()` 方法就是输出字符串。

`repaint()` 方法的功能是：程序首先清除用 `paint()` 方法以前所画的内容，然后再调用 `paint()` 方法。

【范例 21-3】 Applet 程序方法使用范例 (ch21\HelloAppletDemo)。

❶ 打开编辑器，输入以下代码。

```
01 package test ;
02 import java.awt.Graphics;
03 import java.applet.Applet;
04 public class HelloAppletDemo extends Applet
05 {
06     String mystring="";
07     public void paint(Graphics g)
08     {
09         g.drawString(mystring,5,30);           //输出字符串
10     }
11     public void init()
12     {
13         mystring=mystring+"正在初始化……";
14         repaint();
15     }
16     public void start()
17     {
18         mystring=mystring+"正在开始执行程序……";
19         repaint();
20     }
21     public void stop()
22     {
23         mystring=mystring+"正在停止执行程序……";
24         repaint();
25     }
26     public void destory()
27     {
28         mystring=mystring+"正在收回资源……";
29         repaint();
30     }
31 }
```

❷ 输入以下 hello.html 代码。

```
<HTML>
<HEAD>
<TITLE> Applet 程序 </TITLE>
<BODY>
```



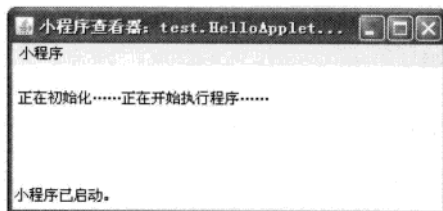
```

    <APPLET CODE="test.HelloAppletDemo" WIDTH="300" HEIGHT="100">
    </APPLET>
  </BODY>
</HTML>

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

由此可以得到 Applet 程序运行的过程：在浏览器打开网页时，会调用 Applet 对象的 init() 方法，接着是 start() 方法，在离开此网页时，会调用 Applet 对象的 stop() 方法，接着是 destroy() 方法。

21.3 在 HTML 中嵌入 Applet 程序

本节主要介绍在 HTML 中嵌入 Applet 程序的方法。下面先介绍一下 HTML 代码的基本结构。

21.3.1 HTML 代码的基本结构

在 WWW 中，每一个显示单位称为一个网页，该网页是由 HTML 语言（HyperText Markup Language，超文本标记语言）编写而成的。HTML 是一种分层语言，各种标记均成对出现，用“<”“>”括起来，开始和结束标记的区别在于结束标记以“/”开头，标记字母忽略大小写。每个页面都必须包含相同的整体结构，如下所示。

```

<HTML>
<HEAD> .....
  <TITLE> ..... </TITLE>
</HEAD>
  <BODY> .....</BODY>
</HTML>

```

- (1) HTML 标记是最外层的标记，表示整个文档的开始和结束。
- (2) HEAD 标记是第 2 层，用于把与文档有关的信息与文档主体分开，相当于文档的头部。
- (3) TITLE 标记包含于 HEAD 内，向用户提示文档内包含的信息类型，并且为其页面提供一个描述性的标题。
- (4) BODY 标记表示文档的主体部分。

HTML 语言还包括许多其他的标记。由于篇幅有限，这里只列举了几个，有兴趣的读者可查阅相关资料。

21.3.2 Applet 标记

在<APPLET>标记的完整语法中可以有若干个属性，其中必需的属性是 CODE、WIDTH 和 HEIGHT，其余的均为可选项。<APPLET>标记的属性应该出现在<APPLET>和</APPLET>之间。

下面是<APPLET>标记所具有的属性。

1. CODEBASE = "codebaseURL"

可选属性，它指定 Java 字节代码的路径或 URL。如果未指定该属性，则将使用与.html 文档相同的目录。CODEBASE 的主要用途是告诉 Java 浏览器到哪里寻找在当前目录中没有显示的字节代码文件 (.class)。

2. ARCHIVE = "archiveList"

可选属性，它描述一个或多个包含有要“预加载”的类或其他资源的文档。archiveList 中的文档用“,”分隔。在 JAVA 2 中，具有相同 CODEBASE 的多个 Applet 程序共享一个 ClassLoader 实例。有些客户机代码使用它来实现 Applet 程序间的通信。从安全方面的因素考虑，Applet 程序的类加载器只能从所启动的那个 CODEBASE 中读取信息，这意味着 archiveList 中的文档位于和 codebaseURL 相同的目录中或其子目录中。

3. CODE = "AppletFile"

必需属性，它提供包含 Applet 类的经编译后的 Applet 小程序。AppletFile 是一个已经编译后的 Java Applet 小程序，即扩展名为.class 的文件。在<APPLET>标记中，CODE 和 OBJECT 属性必须有一个存在。

4. OBJECT = "serialiaedApplet"

可选属性，它给出包含 Applet 程序序列化表示的文件名。此时，Applet 程序中的 init()方法将不会被调用，但是其 start()方法将会被调用。由于该属性有一些严格规定的特性，所以不常使用。

5. ALT= "alternateText"

可选属性，它指定在浏览器能识别<APPLET>标记但不能运行 Java Applet 程序时显示的正文内容。

6. NAME= "AppletInstanceName"

可选属性，它用来为 Applet 程序指定一个符号名，该符号名在相同页的不同 Applet 程序之间通信时使用。

7. WIDTH="pixels" HEIGHT = "pixels"

两个必需属性，它们提供 Applet 程序显示区域的初始宽度和高度（单位为像素），但不包括 Applet 程序中各种方法的任何显示窗口或对话框。

8. ALIGN="alignment"

可选属性，它指定 Applet 程序执行结果的对齐方式。该属性的可能值与 IMG 标记相同，即

包括 LEFT、RIGHT、TOP、TEXTTOP、MIDDLE、ABSMIDDLE、BASELINE、BOTTOM 和 ABSBOTTOM 等。

9. VSPACE="pixels" HSPACE="pixels"

两个可选属性，它们指定 Applet 程序执行结果的显示区上下 (VSPACE) 和两边 (HSPACE) 的像素数，对它们的处理方式与 IMG 标记的 VSPACE 和 HSPACE 属性相同。

10. <PARAM NAME="AppletAttribute 1" VALUE="value">

<PARAM NAME= AppletAttribute 2 VALUE = value>

可选属性，它指定给 Applet 程序传递参数的名字和数据。在 Applet 程序中使用 getParameter() 方法可以得到这些参数值。

在 HTML 中使用 <APPLET> 标记的大部分属性是比较容易理解的，唯有传递参数的属性复杂一些，因为它实现了从 Web 页面到 Applet 程序的通信。

21.3.3 在 HTML 中传递 Applet 程序使用的参数

为了使 Applet 程序更具灵活性，需要在小程序中设置一些未知参数，以接受来自 Web 页面的信息。即在 HTML 中需要传递参数给 Applet 程序。在 HTML 中传递 Applet 程序使用的参数，可以使用 <APPLET> 标记的属性 <PARAM> 来实现，在 Applet 程序中可以使用 String getParameter(String name) 方法得到 HTML 中 <APPLET> 标记的 PARAM 属性传递的参数值，该方法可以在任何地方被调用，但是建议在 init() 方法中使用。需要注意的是：参数名是区分大小写的。

【范例 21-4】 在 HTML 中传递 Applet 程序参数 (ch21\AcceptParam)。

```

01 package test ;
02 import java.awt.Graphics;
03 import java.applet.Applet;
04 public class AcceptParam extends Applet
05 {
06     String tempString,score;
07     public void init()
08     {
09         // 得到 web 页中的 str 参数的值
10         tempString = getParameter("str");
11         if(tempString.equals("及格"))                // 如果字符串等于“及格”
12             score = "60-70";
13         else if(tempString.equals("中"))              // 如果字符串等于“中”
14             score = "70-80";
15         else if(tempString.equals("良"))              // 如果字符串等于“良”
16             score = "80-90";

```

```

17         else if(tempString.equals("优"))           // 如果字符串等于“优”
18             score = "90-100";
19         else
20             score = "0-60";
21     }
22     public void paint(Graphics g)
23     {
24         g.drawString(score,10,25);                 // 输出分数段
25     }
26 }

```

下面编写 HTML 文件，在 HTML 源文件中使用 PARAM 属性来指定 APPLET 程序欲使用的参数值。

【范例 21-5】 在 HTML 中传递 Applet 程序参数 HTML 文件的编写（ch21\AcceptParam）。

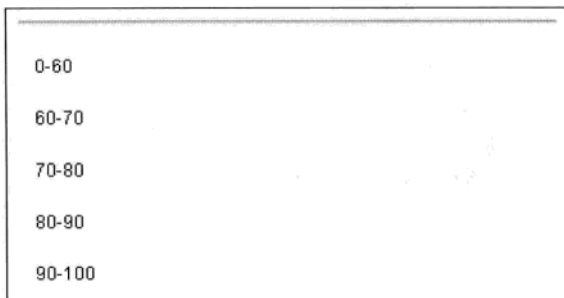
```

<HTML>
  <HEAD><TITLE>在 HTML 中传递 Applet 使用的字符串参数</TITLE></HEAD>
  <HR>
  <BODY>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
      <PARAM NAME="str" VALUE="差">
    </APPLET>
    <BR>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
      <PARAM NAME="str" VALUE="及格">
    </APPLET>
    <BR>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
      <PARAM NAME="str" VALUE="中">
    </APPLET>
    <BR>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
      <PARAM NAME="str" VALUE="良">
    </APPLET>
    <BR>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
      <PARAM NAME="str" VALUE="优">
    </APPLET>
  </BODY>
</HTML>

```

【运行结果】

保存并运行程序，结果如图所示。



随着 Java 技术的不断发展，Applet 程序的使用已经逐渐减少。在这里读者只需要了解 Applet 程序的基本组成，以及如何使用 Applet 程序即可。

21.4 练一练

一、填空题

1. Applet 程序中必须有一个是_____类的子类。
2. Applet 程序不能单独运行，必须通过_____调入后才能执行，以实现其功能。
3. Applet 的生命周期是指_____。
4. <APPLET>标记的属性应该出现在_____和_____之间。

二、简答题

简述 HTML 语言的结构。

21.5 跟我上机

编写一个 Java Applet 程序，运行后显示“这是我的 Java Applet 程序!”。

第 22 章

Java 网络程序设计



本章视频教学录像：39 分钟

现代人的生活已经越来越离不开网络，网络程序设计是Java程序设计的一个重要组成部分，使用Java可以轻松地开发出各种类型的网络程序。本章介绍网络程序设计的相关基本概念，内容包括Socket介绍、Socket程序设计、TCP程序的实现、编写简单的TCP程序，以及UDP程序的实现等。

本章要点（已掌握的在方框中打勾）

- ☐ 熟悉 Socket 相关概念
- ☐ 掌握 Socket 程序的设计
- ☐ 熟悉 TCP 程序的实现方法
- ☐ 掌握简单的 TCP 程序设计
- ☐ 了解 UDP 程序的实现方法



22.1 Socket 介绍

Socket 是网络上运行的两个程序间双向通信的一端，它既可以接受请求，也可以发送请求，利用它可以较为方便地实现网络上数据的传递。在 Java 中，有专门的 Socket 类来处理用户的请求和响应。利用 Socket 类的方法，就可以实现两台计算机之间的通信。本章介绍在 Java 中如何利用 Socket 进行网络编程。

在 Java 中，可以将 Socket 理解为客户端或者服务器端的一个特殊的对象，这个对象有两个关键的方法，一个是 `getInputStream()` 方法，另一个是 `getOutputStream()` 方法。`getInputStream()` 方法用于得到一个输入流，客户端的 Socket 对象上的 `getInputStream()` 方法得到的输入流其实就是从服务器端发回的数据流。`getOutputStream()` 方法用于得到一个输出流，客户端 Socket 对象上的 `getOutputStream()` 方法返回的输出流就是将要发送到服务器端的数据流（其实是一个缓冲区，暂时存储将要发送过去的数据）。

Socket 有两种主要的操作方式：面向连接的和无连接的。面向连接的 socket 操作就像一部电话，它们必须建立一个连接和一个呼叫。所有的事情到达时的顺序与它们出发时的顺序是一样的。无连接的 socket 操作就像是一个邮件投递，没有什么保证，多个邮件可能到达时的顺序与出发时的顺序不一样。

到底用哪一种模式应根据应用程序的需要决定。如果可靠性更重要的话，用面向连接的操作会好一些。比如文件服务器需要保证数据的正确性和有序性，如果一些数据丢失了，系统的有效性将会失去。一些服务器，比如间歇性地发送一些数据块，如果数据丢了的话，服务器并不想要再重新发一次，因为当数据到达的时候，它可能已经过时了。确保数据的有序性和正确性需要的额外操作会消耗内存，额外的费用将会降低系统的回应速率。

无连接的操作使用数据报协议。一个数据报是一个独立的单元，它包含了所有的这次投递的信息。可以把它想象成一个信封，它有目的地址和要发送的内容。这个模式下的 socket 不需要连接一个目的地的 socket，它只是简单地投出数据报。无连接的操作是快速的和高效的，但是数据安全性不佳。

面向连接的操作使用 TCP 协议。在这个模式下建立的 socket 必须在发送数据之前与目的地的 socket 取得一个连接。一旦连接建立了，socket 就可以使用一个流：打开→读→写→关闭。所有发送的信息都会另一端以同样的顺序被接收。面向连接的操作比无连接的操作效率更低，但是数据的安全性较高。

22.2 Socket 程序

在 Java 中面向连接的操作类有两种形式，分别是客户端和服务端。下面先讨论服务器端。首先建立服务器端程序，此服务器端程序只用于向客户端输出“hello world!”字符串。

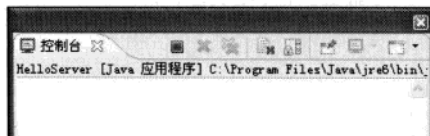
【范例 22-1】 Socket 程序的使用（代码 22-1.java）。

```
01 import java.io.*;
```

```
02 import java.net.*;
03 public class HelloServer
04 {
05     public static void main(String[] args) throws IOException
06     {
07         ServerSocket serversocket=null;
08         PrintWriter out=null;
09         try
10         {
11             // 在下面实例化了一个服务器端的 Socket 连接
12             serversocket=new ServerSocket(9999);
13         }
14         catch(IOException e)
15         {
16             System.err.println("Could not listen on port:9999.");
17             System.exit(1);
18         }
19         Socket clientsocket=null;
20         try
21         {
22             // accept()方法用来监听客户端的连接
23             clientsocket=serversocket.accept();
24         }
25         catch(IOException e)
26         {
27             System.err.println("Accept failed.");
28             System.exit(1);
29         }
30         out=new PrintWriter(clientsocket.getOutputStream(),true);
31         out.println("hello world!");
32         clientsocket.close();
33         serversocket.close();
34     }
35 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码注解】

第 7 行声明了一个 ServerSocket 的对象。

第 8 行声明了一个 PrintWriter 的对象，用于向客户端打印输出。

第 9~18 行实例化 ServerSocket 对象，在 9999 端口进行监听。

第 19 行声明了一个 Socket 对象 clientsocket，此对象用于接收客户端的 Socket 连接。

第 20~29 行通过 ServerSocket 类中的 accept() 方法，接收客户端的 Socket 请求，此方法返回一个客户端的 Socket 请求。

第 30 行通过客户端的 Socket 对象去实例化 PrintWriter 对象，此时 out 对象就具备了向客户端打印信息的能力。

第 31 行调用 println() 方法，将信息打印至客户端。

第 32 行关闭客户端 Socket 连接。

第 33 行关闭服务器端 Socket 连接。

从运行结果可以看到，执行 java HelloServer 之后，程序停在原处不动了，这表示服务器在等待客户端的连接。下面为客户端程序。

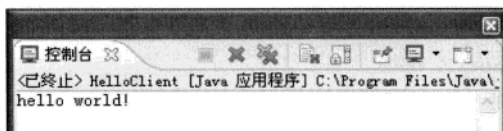
【范例 22-2】 客户端程序编写（代码 22-2.java）。

```
01  import java.io.*;
02  import java.net.*;
03  public class HelloClient
04  {
05      public static void main(String[] args) throws IOException
06      {
07          Socket hellosocket=null;
08          BufferedReader in=null;
09          // 下面这段程序，用来将输入输出流与 socket 关联
10          try
11          {
12              hellosocket=new Socket("localhost",9999);
13              in=new BufferedReader(
14                  new InputStreamReader(hellosocket.getInputStream()));
15          }
16          catch(UnknownHostException e)
17          {
18              System.err.println("Don't know about host:localhost!");
19              System.exit(1);
20          }
21          catch(IOException e)
22          {
23              System.err.println("Couldn't get I/O for the connection.");
24              System.exit(1);
25          }
26      }
27  }
```

```
24     }
25     System.out.println(in.readLine());
26     in.close();
27     hellosocket.close();
28 }
29 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 7 行声明了一个 Socket 的对象 hellosocket。

第 8 行声明了一个 BufferedReader 的对象 in，此对象用于读取服务器端发送过来的数据。

第 12 行实例化 hellosocket 对象，此连接在本机的 9999 端口上监听。

第 13 行通过 hellosocket 对象实例化 BufferedReader 对象。

第 25 行等待服务器端发送过来的信息并打印。

第 26 行关闭 BufferedReader。

第 27 行关闭 Socket 对象。

下面再来看一个 Socket 的经典范例——Echo 程序，读者可以自行分析。

【范例 22-3】 Echo 服务器端程序编写（代码 22-3.java）。

```
01 import java.io.*;
02 import java.net.*;
03 public class EchoServer
04 {
05     public static void main(String[] args) throws IOException
06     {
07         ServerSocket serverSocket = null;
08         PrintWriter out = null;
09         BufferedReader in = null;
10         try
11         {
12             // 实例化监听端口
13             serverSocket = new ServerSocket(1111);
14         }
15         catch (IOException e)
16         {

```



```
17         System.err.println("Could not listen on port: 1111.");
18         System.exit(1);
19     }
20     Socket incoming = null;
21     while(true)
22     {
23         incoming = serverSocket.accept();
24         out = new PrintWriter(incoming.getOutputStream(), true);
25         // 将字节流放入字符流缓冲之中
26         in = new BufferedReader(
27             new InputStreamReader(incoming.getInputStream()));
28         // 提示信息
29         out.println("Hello! ...");
30         out.println("Enter BYE to exit");
31         out.flush();
32         // 在没有异常的情况下不断循环
33         while(true)
34         {
35             // 只有当用户输入数据的时候才返回数据内容
36             String str = in.readLine();
37             // 当用户连接断掉时会返回空值 null
38             if(str == null)
39             {
40                 // 退出循环
41                 break;
42             }
43             else
44             {
45                 // 对用户输入字符串加前缀 Echo:，将此信息打印到客户端
46                 out.println("Echo: "+str);
47                 out.flush();
48                 // 退出命令，equalsIgnoreCase()是不区分大小写的比较
49                 if(str.trim().equalsIgnoreCase("BYE"))
50                 {
51                     break;
52                 }
53             }
54         }
55         // 收尾工作
56         out.close();
57         in.close();
58         incoming.close();
59         serverSocket.close();
60     }
61 }
```

```
57     }  
58 }  
59 }
```

【范例 22-4】 Echo 客户端程序编写（代码 22-4.java）。

```
01 import java.io.*;  
02 import java.net.*;  
03 // 客户端程序  
04 public class EchoClient  
05 {  
06     public static void main(String[] args) throws IOException  
07     {  
08         Socket echoSocket = null;  
09         PrintWriter out = null;  
10         BufferedReader in = null;  
11         try  
12         {  
13             echoSocket = new Socket ( "localhost", 1111);  
14             out = new PrintWriter(echoSocket.getOutputStream(), true);  
15             in = new BufferedReader(  
16                 new InputStreamReader(echoSocket.getInputStream()));  
17         }  
18         catch (UnknownHostException e)  
19         {  
20             System.err.println("Don't know about host: localhost.");  
21             System.exit(1);  
22         }  
23         System.out.println(in.readLine());  
24         System.out.println(in.readLine());  
25         BufferedReader stdIn =  
26             new BufferedReader(new InputStreamReader(System.in));  
27         String userInput;  
28         // 将客户端 Socket 输入流输出到标准输出  
29         while ((userInput = stdIn.readLine()) != null)  
30         {  
31             out.println(userInput);  
32             System.out.println(in.readLine());  
33         }  
34         out.close();  
35         in.close();  
36     }  
37 }
```

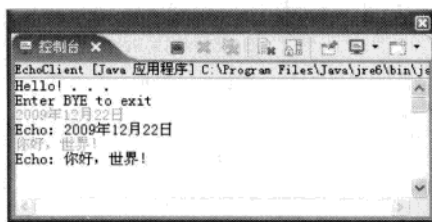
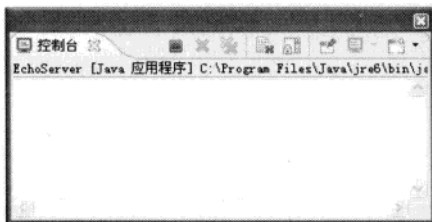
```

34         echoSocket.close();
35     }
36 }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

运行上面的程序可以看到，无论客户端输入什么，服务器端都会对数据进行回显，输入“bye”之后程序会退出。

但细心的读者可能会发现，此程序只能允许一个客户端进行操作，即其他客户端程序无法再进行 Socket 连接，那该如何去解决这个问题呢？读者应该还记得之前讲解过的多线程的概念，只要在服务器端实现多线程，那么服务器就可以同时处理多个客户端请求。下面的代码是改进后的 EchoServer 程序。

【范例 22-5】 EchoServer 程序的改进（代码 22-5.java）。

```

01  import java.net.*;
02  import java.io.*;
03  public class EchoMultiServerThread extends Thread
04  {
05      private Socket socket = null;
06      public EchoMultiServerThread(Socket socket)
07      {
08          super("EchoMultiServerThread");
09          // 声明一个 socket 对象
10          this.socket = socket;
11      }
12      public void run()
13      {
14          try
15          {
16              PrintWriter out = null;
17              BufferedReader in = null;
18              out = new PrintWriter(socket.getOutputStream(), true);

```



```
19         in = new BufferedReader(  
                new InputStreamReader(socket.getInputStream()));  
20         out.println("Hello! . . . ");  
21         out.println("Enter BYE to exit");  
22         out.flush();  
23         while(true)  
24         {  
25             String str = in.readLine();  
26             if(str == null)  
27             {  
28                 break;  
29             }  
30             else  
31             {  
32                 out.println("Echo: "+str);  
33                 out.flush();  
34                 if(str.trim().equalsIgnoreCase("BYE"))  
35                     break;  
36             }  
37         }  
38         out.close();  
39         in.close();  
40         socket.close();  
41     }  
42     catch (IOException e)  
43     {  
44         e.printStackTrace();  
45     }  
46 }  
47 }
```

【范例 22-6】 多线程的服务器端程序编写（代码 22-6.java）。

```
01 import java.io.*;  
02 import java.net.*;  
03     // 多线程的服务器端程序  
04 public class EchoServerThread  
05 {  
06     public static void main(String[] args) throws IOException  
07     {  
08         // 声明一个 serverSocket
```

```

09      ServerSocket serverSocket = null;
10      // 声明一个监听标识
11      boolean listening = true;
12      try
13      {
14          serverSocket = new ServerSocket(1111);
15      }
16      catch (IOException e)
17      {
18          System.err.println("Could not listen on port: 1111.");
19          System.exit(1);
20      }
21      // 如果处于监听状态则开启一个线程
22      while(listening)
23      {
24          // 实例化一个服务端的 socket 与请求 socket 建立连接
25          new EchoMultiServerThread(serverSocket.accept()).start();
26      }
27      // 将 serverSocket 的关闭操作放在循环外
28      // 只有当监听状态为 false 时, 服务才关闭
29      serverSocket.close();
30  }
31  }

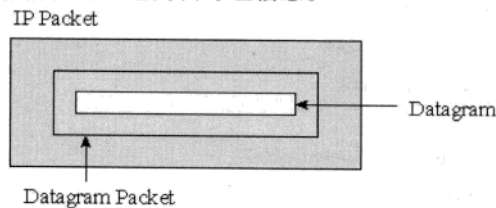
```

【范例分析】

运行上面的服务器端程序之后, 可以看到服务器可以同时处理多个客户端的 Socket 连接。

22.3 DatagramSocket 程序

使用流套接字的每个连接均需要花费一定的时间, 要减少这种开销, 网络 API 提供了第 2 种套接字: 自寻址套接字 (datagram socket)。自寻址使用 UDP 发送寻址信息 (从客户程序到服务程序或从服务程序到客户程序), 不同的是可以通过自寻址套接字发送多个 IP 信息包, 自寻址信息包含在自寻址包中, 自寻址包又包含在 IP 包内, 这样就将寻址信息长度限制在 60000 字节内。下图显示了位于 IP 包内的自寻址包的自寻址信息。



与 TCP 保证信息到达信息目的地的工作方式不同, UDP 提供了另外一种方法, 如果自寻址信息包没有到达目的地, 那么 UDP 也不会请求发送者重新发送自寻址包。这是因为 UDP 在每一个自寻址包中包含了错误检测信息, 在每个自寻址包到达目的地之后 UDP 只进行简单的错误检查, 如果检测失败, UDP 将抛弃这个自寻址包, 也不会从发送者那里重新请求替代者, 这与通过邮局发送信件相似, 发信人在发信之前不需要与收信人建立连接, 同样也不能保证信件能到达收信人那里。

自寻址套接字工作常用的类包括 `DatagramPacket` 和 `DatagramSocket`。`DatagramPacket` 对象描绘了自寻址包的地址信息, `DatagramSocket` 表示客户程序和服务程序自寻址套接字, 这两个类均位于 `java.net` 包内。

1. DatagramPacket 类

在使用自寻址包之前, 需要首先熟悉 `DatagramPacket` 类, 地址信息和自寻址包以字节数组的方式同时压缩进入这个类创建的对象中。

`DatagramPacket` 有数个构造方法, 即使这些构造方法的形式不同, 但通常情况下它们都有两个共同的参数: `byte [] buffer` 和 `int length`。`buffer` 参数包含了一个对保存自寻址数据包信息的字节数组的引用, `length` 表示字节数组的长度。

最简单的构造方法是 `DatagramPacket(byte [] buffer, int length)`, 这个构造方法确定了自寻址数据包数组和数组的长度, 但没有任何自寻址数据包的地址和端口信息, 这些信息可以在后面通过调用方法 `setAddress(InetAddress addr)` 和 `setPort(int port)` 添加上。

2. DatagramSocket 类

`DatagramSocket` 类在客户端创建自寻址套接字与服务器端进行通信连接, 并发送和接受自寻址套接字。虽然有多个构造方法可供选择, 但创建客户端自寻址套接字最便利的选择是 `DatagramSocket()` 方法, 而服务器端则是 `DatagramSocket(int port)` 方法。如果未能创建自寻址套接字或绑定自寻址套接字到本地端口, 那么这两个方法都将抛出一个 `SocketException` 对象, 一旦程序创建了 `DatagramSocket` 对象, 程序就会分别调用 `send(DatagramPacket dgp)` 和 `receive(DatagramPacket dgp)` 来发送和接收自寻址数据包。

【范例 22-7】 Udp 接收数据范例 (代码 22-7.java)。

```
01 import java.net.*;
02 import java.io.*;
03
04 public class UdpReceive
05 {
06     public static void main(String[] args)
07     {
08         DatagramSocket ds = null;
09         byte[] buf = new byte[1024];
10         DatagramPacket dp = null;
11         try {
12             ds = new DatagramSocket(9000);
13         } catch (SocketException ex) {
```

```

14     }
15     // 创建 DatagramPacket 时, 要求的数据格式是 byte 型数组
16     dp = new DatagramPacket(buf, 1024);
17     try {
18         ds.receive(dp);
19     } catch (IOException ex1) {
20     }
21     /*
22     * 调用 public String(byte[] bytes,int offset,int length)构造方法,
23     * 将 byte 型的数组转换成字符串
24     */
25     String str = new String(dp.getData(), 0, dp.getLength()) + " from "
26         + dp.getAddress().getHostAddress() + ":" + dp.getPort();
27     System.out.println(str);
28     ds.close();
29 }
30 }

```

【范例 22-8】 Udp 发送数据范例 (代码 22-8.java)。

```

01 import java.net.*;
02 import java.io.*;
03 public class UdpSend
04 {
05     public static void main(String[] args)
06     {
07         // 要编写 UDP 网络程序, 首先要用到 java.net.DatagramSocket 类
08         DatagramSocket ds = null;
09         DatagramPacket dp = null;
10         try {
11             ds = new DatagramSocket(3000);
12         }
13         catch (SocketException ex) {
14         }
15         String str = "hello world ";
16         try {
17             dp = new DatagramPacket(str.getBytes(), str.length(), InetAddress
18                 .getByName("localhost"), 9000);
19             // 调用 getByName()方法可以返回一个 InetAddress 类的实例对象
20         } catch (UnknownHostException ex1) {
21         }
22     }
23 }

```

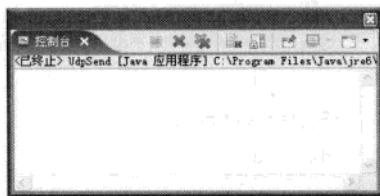
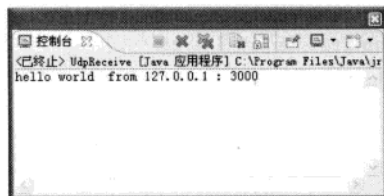
```

22     try {
23         ds.send(dp);
24     } catch (IOException ex2) {
25     }
26     ds.close();
27 }
28 }

```

【运行结果】

保存并运行程序，结果如图所示。



UDP 数据的发送，类似发送寻呼信号，发送者将数据发送出去就不管了，因此是不可靠的数据传输，有可能在发送的过程中数据丢失。就像寻呼机必须先处于开机接收状态才能接收寻呼一样的道理，这里要先运行 UDP 接收程序，再运行 UDP 发送程序，UDP 数据包的接收是过期作废的。因此，前面的接收程序要比发送程序早运行才行。

当 UDP 接收程序运行到 `DatagramSocket.receive()` 方法接收数据时，如果还没有可以接收的数据，在正常情况下，`receive()` 方法将阻塞，一直等到网络上有数据到来，`receive` 才接收该数据并返回。

22.4 网络编程的基本概念

▶ 本节视频教学录像：4 分钟

网络，就是将物理上不在一起的主机进行互联。

在网络上进行通信需要使用协议，常见的通信协议是 TCP 和 UDP。

- TCP：属于可靠的连接，使用三方握手的方式完成连接的确认。
- UDP：属于不可靠的连接。

对于网络程序的开发有两种架构。

- C/S：客户端/服务器端，对于这种程序需要开发两套代码，一套是客户端，另外一套是服务器端，维护也要维护两套。
- B/S：浏览器/服务器，类似于论坛，开发和维护的时候只需要一套代码即可。

22.5 TCP 程序实现

▶ 本节视频教学录像：25 分钟

在 Java 中，所有的网络开发包保存在 `java.net` 包中，在此包中可以使用 `ServerSocket`、`Socket`

类完成服务器和客户端的开发。

22.5.1 简单的 TCP 程序

如果想要开发 TCP 程序，首先要开发服务器端，在服务器端，要使用 `ServerSocket` 进行客户端的连接接收，每一个客户端在程序上都使用 `Socket` 对象表示。

```
package org.lxh.demo.helloserver;
import java.io.OutputStream;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

public class HelloServer {
    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(8888);    //在 8888 端口上开启服务
        Socket client = null;                          //表示连接的客户端
        System.out.println("等待客户端连接...");
        client = server.accept();                       //接受客户端的连接
        OutputStream out = client.getOutputStream();    //得到客户端的输出流
        PrintStream pout = new PrintStream(out);
        pout.println("hello world!!!");
        pout.close();
        out.close();
        client.close();
        server.close();
    }
}
```

如果此时想要进行连接的操作实验，则可通过 `telnet` 命令完成。

此时就完成了了一个服务器程序的开发，此程序执行完一次之后将关闭。那么现在的程序是通过 `telnet` 命令完成的服务器的访问，也可以通过单独的客户端程序编写代码，进行访问。

```
package org.lxh.demo.helloserver;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.Socket;

public class HelloClient {
    public static void main(String[] args) throws Exception {
        Socket client = new Socket("localhost", 8888); //表示连接的主机及端口
        BufferedReader buf = null;
        buf = new BufferedReader(new InputStreamReader(client.getInputStream()));
        String str = buf.readLine();                  //接收回应的内容
    }
}
```

```

        System.out.println("内容是: " + str);
        client.close();
    }
}

```

22.5.2 Echo 程序

下面通过 `ServerSocket` 和 `Socket` 类完成一个简单的 echo 程序，echo 表示回应程序，输入的内容发送到服务器端之后，在前面加上“ECHO”的字符串再返回。

对于服务器端而言，客户端的输出流是服务器端的输入流，服务器端的输出流是客户端的输入流。

```

package org.lxh.demo.echoserver;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(8888);           //在 8888 端口上开启服务
        Socket client = null;                                   //表示连接的客户端
        boolean flag = true;
        while (flag) {
            System.out.println("等待客户端连接...");
            client = server.accept();                             //接受客户端的连接
            BufferedReader buf = new BufferedReader(new InputStreamReader(
                client.getInputStream()));
            PrintStream pout = new PrintStream(client.getOutputStream()); //得到客户端的输出流
            boolean temp = true;
            while (temp) {                                       //循环接收用户输入的内容并回应
                String str = buf.readLine();
                if (str == null || "".equals(str)) {
                    temp = false;
                    break;
                }
                if ("bye".equals(str)) {
                    temp = false;
                    break;
                }
                pout.println("ECHO:" + str);                     //回送信息
            }
        }
    }
}

```



```

    }
    pout.close();
    client.close();
}
server.close();
}
}

```

那么,此时客户端的操作也应该是一样的,应该准备好输入流和输出流,而且现在的代码中存在着输入的操作,所以要使用键盘输入信息。

```

package org.lxh.demo.echoserver;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;
public class EchoClient {
    public static void main(String[] args) throws Exception {
        Socket client = new Socket("localhost", 8888); //表示连接的主机及端口
        BufferedReader input = new BufferedReader(new InputStreamReader(
            System.in));
        BufferedReader buf = null;
        buf = new BufferedReader(new InputStreamReader(client.getInputStream()));
        PrintStream out = new PrintStream(client.getOutputStream());
        boolean flag = true;
        while (flag) {
            System.out.print("请输入要发送的内容: ");
            String str = input.readLine(); //接收回应的内容
            if (str == null || "".equals(str)) {
                flag = false;
                break;
            }
            if ("bye".equals(str)) {
                flag = false;
                break;
            }
            out.println(str);
            System.out.println(buf.readLine());
        }
        client.close();
    }
}

```

}

但是，以上的程序只适合于一个线程使用。如果有多个用户，则肯定无法同时连接，这就是传统单线程的操作。如果想要实现多线程，那么每一个客户端都应该使用线程表示出来。

22.5.3 加入多线程

每一个客户端都使用一个线程对象表示。

```
package org.lxx.demo.echothreadserver;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;
public class EchoThread implements Runnable {
    private Socket client;
    public EchoThread(Socket client) {
        this.client = client;
    }
    public void run() {
        try {
            BufferedReader buf = new BufferedReader(new InputStreamReader(
                client.getInputStream()));
            PrintStream pout = new PrintStream(client.getOutputStream()); // 得到客户端的输出流
            boolean temp = true;
            while (temp) { // 循环接收用户输入的内容并回应
                String str = buf.readLine();
                if (str == null || "".equals(str)) {
                    temp = false;
                    break;
                }
                if ("bye".equals(str)) {
                    temp = false;
                    break;
                }
                pout.println("ECHO:" + str); // 回送信息
            }
            pout.close();
            client.close();
        } catch (Exception e) {
        }
    }
}
```

```

    }
}

```

之后的服务器端在每次接收到客户端的请求之后，直接启动一个新的线程即可。

```

package org.lxh.demo.echothreadserver;
import java.net.ServerSocket;
public class EchoServer {
    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(8888);    // 在 8888 端口上开启服务
        boolean flag = true;
        while (flag) {
            System.out.println("等待客户端连接...");
            new Thread(new EchoThread(server.accept())).start();
        }
        server.close();
    }
}

```

通过此代码可以清楚地知道一个服务器应该实现多线程。

22.6 UDP 程序实现

▶ 本节视频教学录像：10 分钟

UDP 程序使用数据报的形式出现，需要使用以下两个类。

- 数据报的内容：DatagramPacket。
- 发送和接收数据报：DatagramSocket。

在开发 TCP 程序的时候，是先要有服务器端，之后再进行客户端的开发。而 UDP 要运行的时候，则应该先运行客户端，之后再运行服务器端。

1. 客户端

```

package org.lxh.demo.udpdemo;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
public class UDPReceive {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket(3000);
        // 客户端在 3000 端口等待接收服务器发来的信息
        DatagramPacket pack = new DatagramPacket(new byte[1024], 1024);
        // 开辟 1024 的空间
        socket.receive(pack);
    }
}

```

```
        System.out.println("接收到的内容是: " + new String(pack.getData()));  
    }  
}
```

2. 服务器端

```
package org.lxh.demo.udpdemo;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetAddress;  
public class UDPSend {  
    public static void main(String[] args) throws Exception {  
        DatagramSocket socket = new DatagramSocket(9000); // 服务器端在 9000 端口等待接收服务器发  
        来的信息  
        String str = "hello world!!!";  
        DatagramPacket pack = new DatagramPacket(str.getBytes(), 0, str  
        length(), InetAddress.getLocalHost(), 3000); // 向 3000 端口发送消息  
        socket.send(pack);  
    }  
}
```

22.7 练一练

一、填空题

1. 在 Java 中, Socket 对象的两个关键方法是_____和_____。
2. Socket 的两种主要的操作方式是_____和_____。
3. 互联网常见的通信协议有_____协议和_____协议。
- 4 在 Java 中, 所有的网络开发包保存在_____包中。

二、简答题

简述网络程序开发的两种构架。

22.8 跟我上机

编写一个服务器端/客户端程序, 对客户端输入的字符串, 服务器端以“客户端:”开头再返回。

第 23 章

Java数据库编程



本章视频教学录像：1 小时 36 分钟

在已有的Java函数库中,有一组专门处理数据库连接的API:JDBC(Java DataBase Connective)。本章重点讲解来自Java的数据精华——JDBC。熟练地使用这组API能使我们同数据库的沟通更加密切。

本章要点(已掌握的在方框中打勾)

- ☐ 掌握 Java 数据库连接的基本概念
- ☐ 掌握 Java 数据库的连接方法
- ☐ 熟悉 Java 数据库连接的相关类
- ☐ 了解批处理的相关概念
- ☐ 了解事务处理的相关概念
- ☐ 熟悉 MySQL 数据库的使用

23.1 数据库连接的基本概念

本节视频教学录像：4 分钟

JDBC (Java Database Connective), 即 Java 数据库连接, 是一组专门负责连接并操作数据库的标准。JDBC 目前的版本已更新到 3.0。JDBC 的出现, 给使用 Java 的程序设计师提供了单一的数据访问方式, 同时也使得程序设计师利用这组 API, 能有效地访问各种形式的数据, 包括关系型数据库、表格到一般的文本文件。另一方面, 在整个 JDBC 中大量提供的实际上是接口, 通过接口, JDBC 规范了不同数据源的相同访问方式, 并针对各个不同的数据库供应商, 凡是想要使用 Java 进行数据库的开发, 则肯定要对 JDBC 标准有所支持。

JDBC 为程序设计师还有数据库供应商提供了唯一的 API 接口, 让程序设计师和数据库供应商遵循相同的方式进行数据访问。所以, 程序设计师只需要知道如何使用 JDBC 便可以轻松地进行数据访问, 而数据库供应商也可同样专心于其所要实现的, 不必担心其他部分。

就好像程序设计人员和数据库供应商是两座隔海相望的城市, 而 JDBC 的接口则是连接两座城市的跨海大桥。JDBC 有效而和谐地将两者连接起来, 各项分工明确, 程序也会有条不紊地进行。

JDBC 能够让我们完成许多事情。

- (1) 完成数据库的连接创建。
- (2) 传送 SQL 命令给数据库, 完成数据库操作及数据表。
- (3) 接受和处理数据库所执行的结果。

JDBC 在使用中常见的有以下 3 类。

1. JDBC-ODBC 桥连接 (JDBC-ODBC Bridge)

本套连接是 SUN 在 JDK 的开发包中提供的最标准的一套 JDBC 操作类库。要将 JDBC 与数据库之间进行有效的连接访问, 中间要经过一个 ODBC 的连接, 但这意味着整体的性能将会降低, 当读者接触的项目很大或者是用户很多的时候, 维护 ODBC 所需要的工作量庞大而繁杂, 需要在 JDBC 于 ODBC 之前做数据之间的传递与转换, 容易造成性能的丢失或遗漏。所以在开发中是绝对不会去使用 JDBC-ODBC 的连接方式的。但 ODBC 连接简单易学, 所以初学者学习 JDBC 时应该从 ODBC 开始, 并且有些公司在用户计算机上都设置有 ODBC 连接, 所以不需要再做此类设置。

2. JDBC 连接

使用各个数据库提供商给定的数据库驱动程序, 完成 JDBC 的开发, 这个时候需要在 classpath 中配置数据库的驱动程序。此种数据连接方式在性能上比 JDBC-ODBC 桥连接要好很多。Java 是利用本地的函数库与数据库驱动程序的函数库沟通, 在效率上能够大大提升。但同样, 在进行数据库的连接时, 用户必须掌握有 JDBC 的驱动程序以及数据库驱动程序的函数库, 而且不同的数据库拥有多个不同的驱动程序。在进行数据维护时, 工作量是很大的。

3. JDBC 网络连接

此种连接方式主要使用网络连接数据库, 这就要求驱动程序必须有一个中间层服务器 (middleware server)。用户与数据库沟通时会通过此中间层服务器与数据库连接。而且这种连接方式只需要同中间层服务器做出有效连接, 便可以连接上数据库, 所以在更新维护时会大大地减

少工作量。

23.2 使用数据库的准备工作

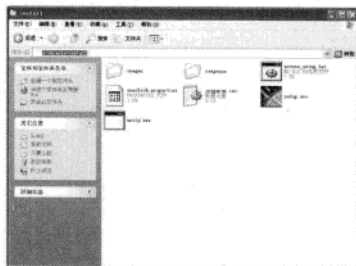
▶ 本节视频教学录像：12 分钟

JDBC 的准备工作非常重要，首先要完成一般的数据库安装工作，这里使用的是 Oracle 数据库。

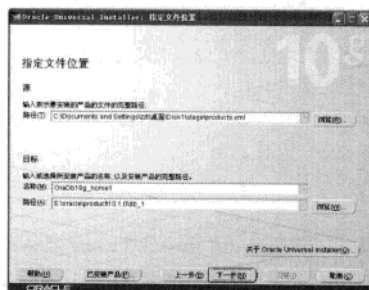
23.2.1 Oracle 数据库的安装

本次进行连接的是 Oracle 数据库。既然是 Oracle 数据库，则肯定在 oracle 中提供了专门的 JDBC 驱动程序。下面介绍 Oracle 的安装方法。

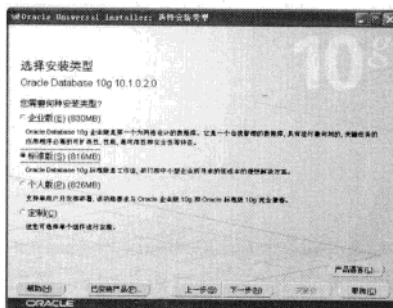
首先打开 Oracle 安装目录下的安装文件夹，双击“setup.exe”文件。



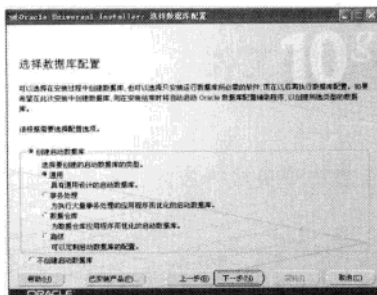
随即出现安装信息提示，按照提示单击【下一步】按钮，进行安装的源和目标文件地点设置。



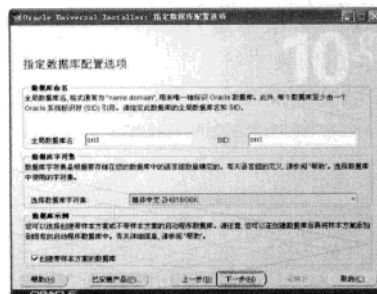
单击【下一步】按钮，随即出现选择安装类型的信息提示，Oracle 分为企业版、标准版和个人版等 3 个版本。本次预先安装标准版本，选中【标准版】单选按钮，然后单击【下一步】按钮。



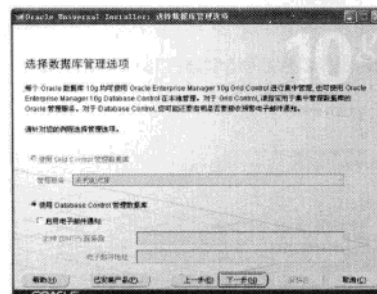
这时需要进行选择数据库的配置，单击【创建启动数据库】，然后单击【下一步】按钮。



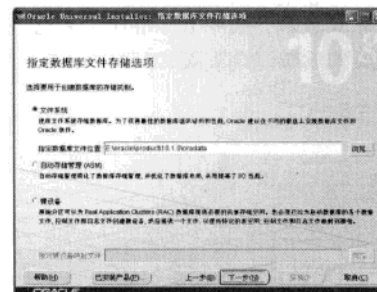
接下来进行数据库配置设置，输入【全局数据库名】和【SID】，然后单击【下一步】按钮。



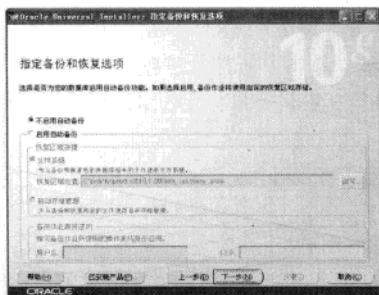
选择数据库管理选项，默认的方式为【使用 Database Control 管理数据库】，然后单击【下一步】按钮。



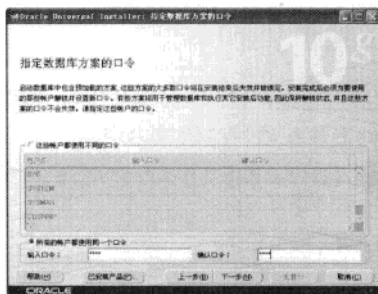
设置数据库文件存储选项，是指设置用户的数据文件存储位置，将自己的目标文件地址键入地址栏之后单击【下一步】按钮。



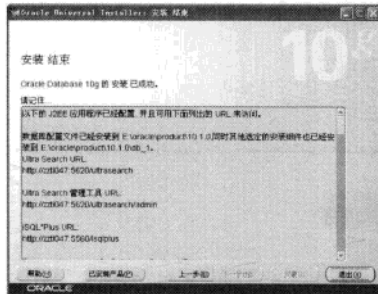
指定备份或恢复选项，选择默认的【不启用自动备份】，然后单击【下一步】按钮。



设置数据库方案的口令, 如果需要使用统一口令, 在下一栏中设置好口令之后, 为了方便起见, 口令统一设置为“java”。记住, 一般 Oracle 默认的用户名称为“system”。安装完成后可以进行变更或添加, 然后单击【下一步】按钮。



设置好全部的安装信息后, 单击【安装】按钮便可得到安装结束的信息。



安装结束后会弹出设置启动数据库的命令网页, 在上面单击启用并通过之前设置的主机用户名及密码和管理员名称及密码后, 即可完成 Oracle 的安装。

23.2.2 数据库连接驱动程序设置

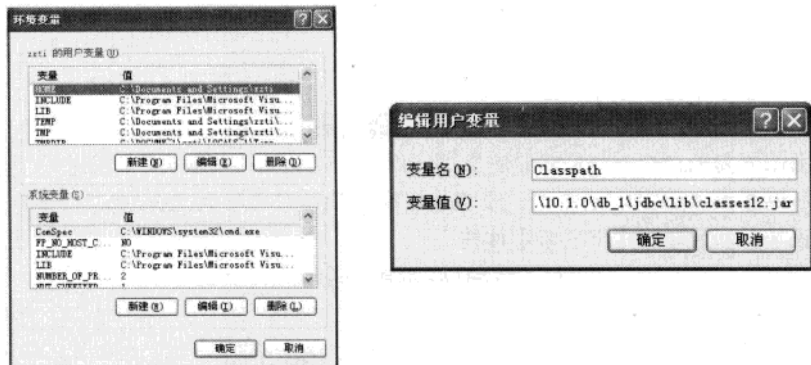
接下来完成对数据库连接的驱动程序的设置。不同的数据库供应商拥有不同数据库的驱动程序。对于 Oracle 这种大型的数据库软件, 都会提供 Java 环境下的数据库驱动程序。首先找到 Oracle 目标目录下的 db_1 文件夹, 可以看到提供给 Java 的驱动程序包 jdbc, 打开 jdbc 中的 lib 文件夹, 其中的 classes12.jar 就是我们需要的驱动程序。

驱动程序路径: E:\oracle\product\10.1.0\db_1\jdbc\lib\classes12.jar

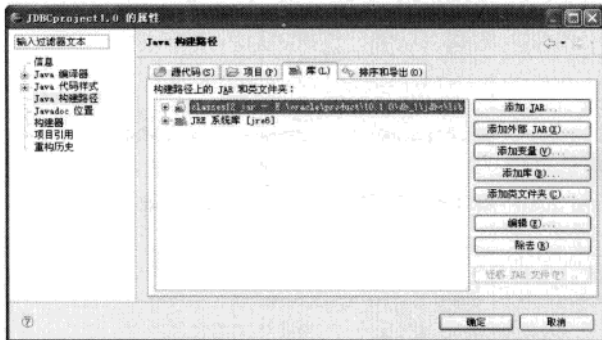
如果现在是直接使用命令行方式进行开发的话, 则需要在属性中增加 classpath。

、 右击【我的电脑】, 在弹出的快捷菜单中选择【属性】命令, 在弹出的【系统属性】对话框

中选择【高级】选项卡，然后单击【环境变量】按钮，在弹出的【环境变量】对话框的【用户变量】框架中选择“Classpath”变量，单击【新建】按钮，在弹出的【编辑用户变量】对话框的【变量名】文本框中输入“Classpath”，在【变量值】文本框中键入刚才得到的驱动程序路径，如图所示。



而如果使用 Eclipse 的话，则直接在项目的属性中增加需要的类库文件即可。



23.2.3 数据库表的准备

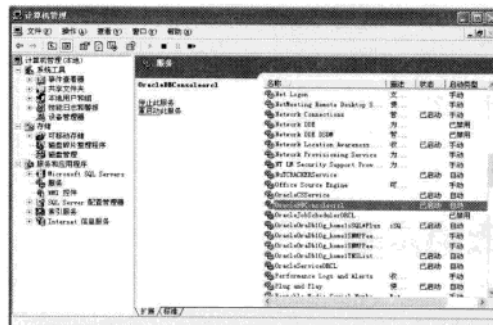
完成了对于 Oracle 数据库的安装，并且为了能够说清 JDBC 的基本操作，应使用以下的数据库表完成操作。

数据库表类型结构如下表所示。

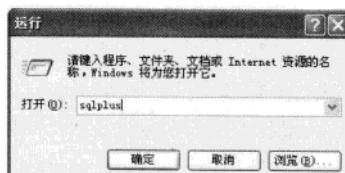
No.	列名称	类型	描述
1	pid	NUMBER	表示人员编号，自动增长，通过 sequence
2	name	VARCHAR2	姓名
3	age	NUMBER	年龄
4	birthday	DATE	生日
5	salary	NUMBER	工资，小数

在 Eclipse 中新建一个名称为 JDBCproject1.0 的包，接下来完成编写数据库的创建脚本。

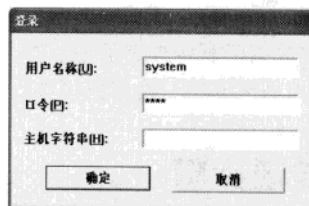
- (1) 新建名称为“数据库创建脚本.sql”的文件。
- (2) 用 SQL 语言新建一个类型为上表类型所示的数据库表，具体语言内容见【范例 23-1】。
- (3) 打开 Oracle 数据库的监听服务和项目服务“OracleJobSchedulerORCL”。



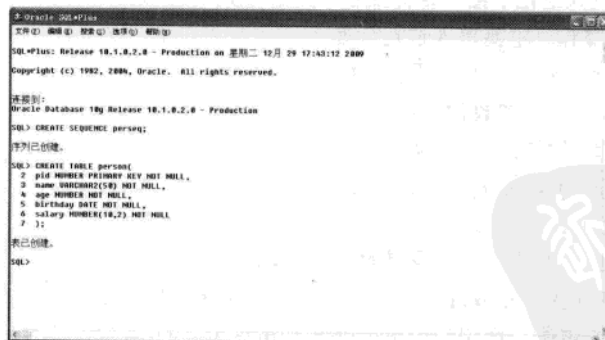
(4) 单击【开始】按钮，选择【运行】命令，弹出【运行】对话框，在【打开】文本框中输入“sqlplus”，单击【确定】按钮。



(5) 输入用户名和刚安装 Oracle 时设置的用户密码，进入到数据库管理后台。



(6) 然后将刚才创建的文件指令输入到控制台中，此时便创建了一个新的名称为 person 的数据表。



【范例 23-1】 建立新序列和新表（代码 23-1.txt）。

```

01 CREATE SEQUENCE perseq; ---自动添加序列
02 CREATE TABLE person( ---新建表 person
03     pid NUMBER PRIMARY KEY NOT NULL,
04     name VARCHAR2(50) NOT NULL,
  
```

```
05     age    NUMBER          NOT NULL ,
06     birthday DATE          NOT NULL ,
07     salary NUMBER(10,2)    NOT NULL
08 );
```

23.3 连接数据库的步骤

▶ 本节视频教学录像：2 分钟

在进行 JDBC 操作的时候，应按照以下的步骤完成。

(1) 加载数据库驱动程序，加载的时候需要将驱动程序配置到 classpath 之中。classpath 是在设置数据库驱动程序时的一个变量。在连接数据库初期的时候，需要在【我的电脑】高级环境变量中设定值。

(2) 连接数据库，可通过 Connection 接口和 DriverManager 类完成。Connection 接口是 JDBC 为数据库的连接所提供的接口，它和 DriverManager 类一样，具有操作连接数据库和控制数据源的作用。

(3) 操作数据库，主要是通过 Statement、PreparedStatement、ResultSet 等 3 个接口完成。这 3 个接口在后面会介绍。

(4) 关闭数据库。在实际开发中数据库资源非常有限，操作完之后必须关闭，否则会造成数据的泄露或丢失，为用户或者开发人员造成不可挽回的损失。

23.4 数据库连接的详细步骤

▶ 本节视频教学录像：8 分钟

完成了数据库的创建之后，接下来需要在 JDBC 中建立与数据库之间的关联操作，也就是完成 Java 数据库的连接操作，主要是通过包含在 Java API 包下的 Class 类中的方法进行。在 JDBC 的操作过程中，进行数据库连接的主要步骤如下。

(1) 通过 Class.forName()加载数据库的驱动程序。首先需要利用来自 Class 类中的静态方法 forName ()，加载需要使用的 Driver 类。

(2) 通过 DriverManager 类进行数据库的连接。成功加载 Driver 类以后，Class.forName () 会向 DriverManager 注册该类，此时则可通过 DriverManager 中的静态方法 getConnection 进行数据库的创建连接。同时，连接的时候需要输入数据库的连接地址、用户名、密码。

(3) 通过 Connection 接口接收连接。当成功进行了数据库的连接之后，getConnection 方法会返回一个 Connection 的对象，而 JDBC 主要就是利用这个 Connection 对象与数据库进行沟通。

(4) 此时输出的是一个对象，表示数据库已经连接上了。

【范例 23-2】 通过 Java 指令进行实际数据库的连接。本例是在用户建立过 Oracle 数据表之后，通过 Java 程序进行数据库的连接（代码 23-2.java）。

```
01 package wf.network.JDBCproject01;
02 import java.sql.Connection;
```

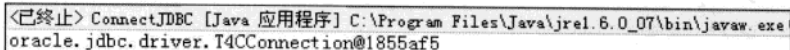
```

03 import java.sql.DriverManager;
04 public class ConnectJDBC {
05     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
06     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
07     // 连接地址是由各个数据库生产商单独提供的, 所以需要单独记住
08     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:orcl";
09     // 连接数据库的用户名
10     public static final String DBUSER = "system";
11     // 连接数据库的密码
12     public static final String DBPASS = "java";
13     public static void main(String[] args) throws Exception {
14         Connection conn = null;           // 表示数据库的连接的对象
15         // 1. 使用 Class 类加载驱动程序
16         Class.forName(DBDRIVER);
17         // 2. 连接数据库
18         conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
19         System.out.println(conn);
20         // 3. 关闭数据库
21         conn.close();
22     }
23 }

```

【运行结果】

保存并运行程序, 结果如图所示。



```

已终止 ConnectJDBC [Java 应用程序] C:\Program Files\Java\jre1.6.0_07\bin\javaw.exe
oracle.jdbc.driver.T4CConnection@1855af5

```

【范例分析】

这个范例特别重要, 需要读者注意。首先创建一个连接 JDBC 的类, 类名定为 ConnectJDBC。在第 6 行中, 定义声明的为数据库驱动程序的地址, 一般在 Java 环境下的数据库驱动程序地址均为此, 所以不需要特别记忆。

重点是第 8 行, 此处声明的是数据库的地址, 也就是所建立数据库服务器的地址。由于数据库提供商所提供的数据库不同, 数据库的连接地址也不尽相同, 所以在 Oracle 中, 这条地址需要特别记忆。就本行代码为例, jdbc:oracle:thin:@ 这行代码为地址头, 一般的 Oracle 数据库地址均由此开始。@之后的 localhost 为本机的 IP 地址, 如果访问其他服务器的地址, 则需要键入第三方服务器的 IP 地址进行连接。1521 为端口号, Oracle 默认的端口号为 1521, 也可登录 Oracle 目录下的 Net Manager 查看。最后的 orcl 为数据库服务器名称。

设定好数据库的驱动程序及链接地址后, 第 10 行则是用户名称, 这里用的是 Oracle 默认携带的用户名 system。第 12 行为用户密码, 也就是当前用户的密码, 这里设置的密码为 java。关于用户名和用户密码, 均在安装 Oracle 初期进行过设置。

第 14~21 行为数据库连接密码，首先声明一个 Connection 对象 conn，对象内容为空，同时，在第 16 行运用 Class.forName () 加载驱动程序，第 18 行通过 getConnection () 方法进行连接，在此方法中分别调用先前声明的数据库地址、用户名和用户密码。

最后在第 21 行关闭数据库。

如果此时输出的为一个对象，则表示连接成功。

23.5 数据维护

▶ 本节视频教学录像：9 分钟

如果想要进行数据库的维护操作，则可使用 Statement 接口，数据库维护主要是进行增加、修改和删除等操作。

23.5.1 增加数据

增加操作要编写增加的 SQL 语句：INSERT。因为涉及到了序列的概念，所以在需要使用的时候需要编写 SQL 语句进行增加操作，在 SQL 语句中直接写上序列的 nextVal 即可。形式为：序列名.nextVal。

【范例 23-3】 为数据表增加数据。这里使用 Statement 接口为数据库中新建立的表进行增加数据操作（代码 23-3.java）。

```
01 package wf.network.updatedemo01;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.Statement;
05 public class StatementDemo01 {
06     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
07     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
08     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
09     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:orcl";
10     // 连接数据库的用户名
11     public static final String DBUSER = "system";
12     // 连接数据库的密码
13     public static final String DBPASS = "java";
14     public static void main(String[] args) throws Exception {
15         Connection conn = null;        // 表示数据库连接的对象
16         Statement stmt = null;         // 表示数据库的更新操作
17         // 1. 使用 Class 类加载驱动程序
18         Class.forName(DBDRIVER);
19         // 2. 连接数据库
20         conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
```



```

21      // 3. Statement 接口需要通过 Connection 接口进行实例化操作
22      stmt = conn.createStatement();
23      // 执行 SQL 语句, 更新数据库
24      stmt.executeUpdate("INSERT INTO person(pid,name,age,birthday,salary)
25      VALUES (perseq.nextval,'张三',30,TO_DATE('1995-02-14','yyyy-mm-dd'),9000.0) 26");
26      // 4. 关闭数据库
27      conn.close();
28  }
29  }

```

【范例分析】

还是同【范例 23-2】, 需要添加的为第 16 行中的 Statement 接口, 设置此接口对象为空, 表示数据库的更新操作。同时, 在第 22 行中执行 SQL 语句, 更新数据库。SQL 语句一般写在 Statement 接口下的 executeUpdate() 方法里, 如第 24 行所示。这时运行程序, 如果没有输出项, 则表示数据库增加成功。

需要注意的是日期格式的数据, 由于 Oracle 的局限性, 这里需要增添 TO_DATE 语句即 DATE 格式数据, 第 25 行中设置的日期格式为年-月-日。

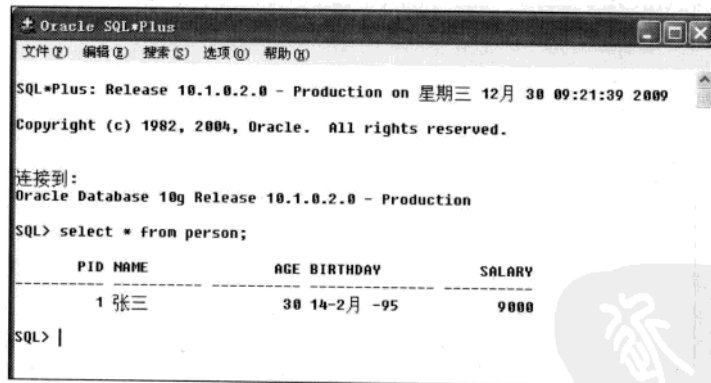
接下来通过 sqlplus 进行查询, 观察输入的信息是否存入到数据库当中。

开始→运行→sqlplus。

输入用户名 system, 密码 java。

进入到 SQL 控制后台, 写入查询的 SQL 语句 “select * from person”。

便可看到自己新增加的内容, 如图所示。



23.5.2 更新数据

数据库的更新操作和增加数据的操作相同, 直接编写 update 语句即可。

【范例 23-4】 更新数据。在其后增加更新数据的 SQL 语句 (代码 23-4.java)。

```

01  package wf.network.updatedemo01;
02  import java.sql.Connection;

```



```
03 import java.sql.DriverManager;
04 import java.sql.Statement;
05 public class StatementDemo02 {
06     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
07     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
08     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
09     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:orcl";
10     // 连接数据库的用户名
11     public static final String DBUSER = "system";
12     // 连接数据库的密码
13     public static final String DBPASS = "java";
14     public static void main(String[] args) throws Exception {
15         Connection conn = null;        // 表示数据库的连接对象
16         Statement stmt = null;         // 表示数据库的更新操作
17         // 1. 使用 Class 类加载驱动程序
18         Class.forName(DBDRIVER);
19         // 2. 连接数据库
20         conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
21         // 3. Statement 接口需要通过 Connection 接口进行实例化操作
22         stmt = conn.createStatement();
23         // 执行 SQL 语句，更新数据库
24         stmt.executeUpdate("UPDATE person SET name='李四',
25                             age=33,birthday=sysdate,salary=8000.0 WHERE pid=1");
26         // 4. 关闭数据库
27         conn.close();
28     }
29 }
```

【范例分析】

SQL 语句是进行数据库连接必不可少的语句，准确而有效地运用 SQL 语句是学好 Java 连接数据库，包括数据库系统的首要操作。

在第 24~25 行，运用更新的 SQL 语句将原先数据库中的“张三”更新为“李四”，这里需要注意的是在更新地址的目标上需要键入该条数据的识别码。

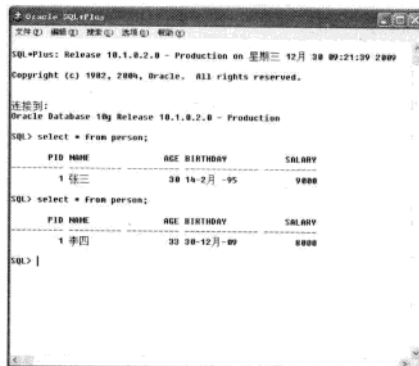
接下来通过 sqlplus 进行查询，观察输入的信息是否存入到数据库当中。

开始→运行→sqlplus。

输入用户名 system，密码 java。

进入到 SQL 控制后台，写入查询的 SQL 语句“select * from person”。

更新完成的数据结果如图所示。



23.5.3 删除数据

数据的增加、更新、删除等都是通过 SQL 语句完成的，所以在修改数据的时候，只需要更改该程序下的 SQL 语句即可实现目的。使用 DELETE 语句可以进行删除数据的操作。

删除数据的时候一般都是按照 id 删除。

【范例 23-5】 删除数据库中的数据信息。可通过修改 SQL 语句删除数据。代码的整体流程和上个范例相似，不同的是 SQL 语句。由于数据库操作有自己独特的一种语言，所以当控制数据库操作时，需要使用 SQL 语言才能达到目的（代码 23-5.java）。

```

01 package wf.network.updatedemo01;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.Statement;
05 public class StatementDemo02 {
06     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
07     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
08     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
09     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:orcl";
10     // 连接数据库的用户名
11     public static final String DBUSER = "system";
12     // 连接数据库的密码
13     public static final String DBPASS = "java";
14     public static void main(String[] args) throws Exception {
15         Connection conn = null;        // 表示数据库的连接对象
16         Statement stmt = null;         // 表示数据库的更新操作
17         // 1. 使用 Class 类加载驱动程序
18         Class.forName(DBDRIVER);
19         // 2. 连接数据库
20         conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
21         // 3. Statement 接口需要通过 Connection 接口进行实例化操作
  
```

```

22      stmt = conn.createStatement();
23      // 执行 SQL 语句, 更新数据库
24      stmt.executeUpdate("DELETE FROM person WHERE pid=4");
25      // 4. 关闭数据库
26      conn.close();
27  }
28  }

```

【范例分析】

通过执行 SQL 语句进行删除操作, 在新建表的同时, 规定有他的 id 号, 也就是所储存的位置号码 pid。由于 pid 具有唯一性, 在做删除操作时需要引用 pid 号码, 也就是调用其位置进行删除。

在第 24 行, 需要执行的 SQL 语句均包含在 Statement 接口的 executeUpdate 方法中。在使用时只需要调用其方法, 而后将 SQL 语句包含在内即可。

应该多添加些表中的内容然后进行删除, 这样结果会更明显。

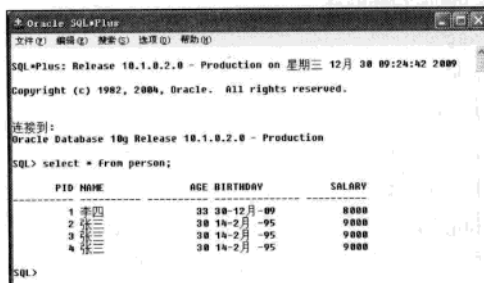
接下来通过 sqlplus 进行查询, 观察输入的信息是否存入到数据库当中。

开始→运行→sqlplus。

输入用户名 system, 密码 java。

进入到 SQL 控制后台, 写入查询的 SQL 语句 “select * from person”。

得到的信息如图所示。



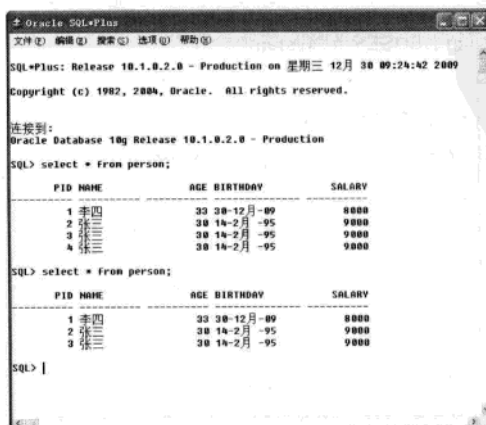
```

SQL> select * from person;

```

PID	NAME	AGE	BIRTHDAY	SALARY
1	李四	33	38-12月-09	8000
2	张三	38	14-2月-95	9000
3	张三	38	14-2月-95	9000
4	张三	38	14-2月-95	9000

运行程序后通过 sqlplus 检查删除结果。



```

SQL> select * from person;

```

PID	NAME	AGE	BIRTHDAY	SALARY
1	李四	33	38-12月-09	8000
2	张三	38	14-2月-95	9000
3	张三	38	14-2月-95	9000

23.6 查询数据库中的内容

▶ 本节视频教学录像：10 分钟

在 Oracle 中，可以通过 SELECT 语句查询数据库中的内容。在 Oracle 中直接查询的时候，可以看到 Oracle 能够将全部的查询结果返回给用户。而对于程序的操作中也是一样，所有的查询结果要返回到程序处输出查看，那么程序就可以通过 ResultSet 接口保存全部的查询结果，通过 Statement 接口中的 executeQuery()方法查询。

查询之后的数据需要分别取出。通过 next()方法找到返回的每一行数据，每一行中各个列的数据需要通过以下方法取得。

例如取得整型：

getInt()

取得字符串：

getString()

取得日期：

getDate()

取得浮点数：

getFloat()

【范例 23-6】 通过 statement 类进行数据库查询操作（代码 23-6.java）。

```
01 package wf.network.resultsetdemo01;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.ResultSet;
05 import java.sql.Statement;
06 import java.util.Date;
07 public class ResultSetDemo01 {
08     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
09     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
10     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
11     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:orcl";
12     // 连接数据库的用户名
13     public static final String DBUSER = "system";
14     // 连接数据库的密码
15     public static final String DBPASS = "java";
16     public static void main(String[] args) throws Exception {
```

```

17 Connection conn = null;           // 表示数据库的连接对象
18 Statement stmt = null;           // 表示数据库的更新操作
19 ResultSet result = null;         // 表示接收数据库的查询结果
20 // 1. 使用 Class 类加载驱动程序
21 Class.forName(DBDRIVER);
22 // 2. 连接数据库
23 conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
24 // 3. Statement 接口需要通过 Connection 接口进行实例化操作
25 stmt = conn.createStatement();
26 // 执行 SQL 语句，查询数据库
27 result = stmt.executeQuery("SELECT pid,name,age,birthday,salary FROM
28 person");
29 while(result.next()){             // 是否有下一行数据
30     int pid = result.getInt("pid");
31     String name = result.getString("name");
32     int age = result.getInt("age");
33     Date birthday = result.getDate("birthday");
34     float salary = result.getFloat("salary");
35     System.out.print("pid = " + pid + " ");
36     System.out.print("name = " + name + " ");
37     System.out.print("age = " + age + " ");
38     System.out.print("birthday = " + birthday + " ");
39     System.out.println("salary = " + salary + " ");
40 }
41 // 4. 关闭数据库
42 result.close();
43 stmt.close();
44 conn.close();
45 }
46 }

```

【运行结果】

保存并运行程序，结果如图所示。

```

问题 Javadoc 声明 控制台 X
<已终止> ResultSetDemo01 [Java 应用程序] C:\Program Files\Java\jre6\bin\javaw.exe
pid=2;name=李四;age=33;birthday=2009-12-27;salary=8000.0;
pid=3;name=张三;age=30;birthday=1995-02-14;salary=9000.0;

```

【代码详解】

第 27~28 行运用 SELECT 语句进行查询工作。需要注意的是：在 SELECT 语句之后出现的是需要查询的明确字段。

第 29 行中的 result.next() 方法用来判断在查询的结果中是否有下一行数据，表示查询信

息是否传入到 result 内部。

第 30~40 行, 由于 java 从数据库中将查询信息调入到 result 内部, 所以在使用 java 程序调出时需要从 result 中调出。此时应用的调出方法是 getXxx()。也就是说, 如果想要得到整型, 则为 getInt(), 依次类推。

第 42~44 行, 将数据库信息打开后需要关闭。

观察 ResultSet 接口的 API 文档。查询取得结果的时候可以通过索引号的方式完成取出数据的操作。

int	getDate(int columnIndex)	以 Java 编程语言中 java.sql.Date 对象的形式获取此 ResultSet 对象的当前行中指定列的值。
int	getDate(int columnIndex, Calendar cal)	以 Java 编程语言中 java.sql.Date 对象的形式获取此 ResultSet 对象的当前行中指定列的值。
int	getDate(String columnLabel)	以 Java 编程语言中的 java.sql.Date 对象的形式获取此 ResultSet 对象的当前行中指定列的值。
int	getDate(String columnLabel, Calendar cal)	以 Java 编程语言中 java.sql.Date 对象的形式获取此 ResultSet 对象的当前行中指定列的值。

【范例 23-7】 用索引号进行数据查询 (代码 23-7.txt)。

```

01 while(result.next()){           // 是否有下一行数据
02     int pid = result.getInt(1);
03     String name = result.getString(2);
04     int age = result.getInt(3);
05     Date birthday = result.getDate(4);
06     float salary = result.getFloat(5);
07     System.out.print("pid = " + pid + "; ");
08     System.out.print("name = " + name + "; ");
09     System.out.print("age = " + age + "; ");
10     System.out.print("birthday = " + birthday + "; ");
11     System.out.println("salary = " + salary + "; ");
12 }

```

【代码详解】

在查询内容里可以直接输入索引号, 不一定必须要用该内容的字段名称。

第 2~6 行, 在查询数据的时候, 可以通过数据所在的索引号查询。

23.7 查询信息实例

本节视频教学录像: 8 分钟

由于之前的程序所有的输入内容都是固定的, 所以在实际操作中, 有很多的可变性就没有被考虑到。那么现在要求可以由用户自己输入需要的内容, 并通过程序把输入的内容加入到数据库中。

【范例 23-8】 通过实例化程序了解控制台输入数据。接下来进行的是当用户自己输入数据时, 需要如何将输入的数据保存到数据库中 (代码 23-8.java)。

```

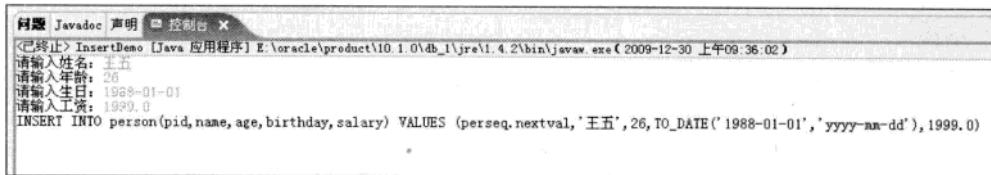
01 package org.lxx.execdemo;

```

```
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.Statement;
05 public class InsertDemo {
06     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
07     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
08     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
09     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:ORCL";
10     // 连接数据库的用户名
11     public static final String DBUSER = "system";
12     // 连接数据库的密码
13     public static final String DBPASS = "java";
14     public static void main(String[] args) throws Exception {
15         Connection conn = null;        // 表示数据库的连接对象
16         Statement stmt = null;          // 表示数据库的更新操作
17         InputData input = new InputData();
18         String name = input.getString("请输入姓名：");
19         int age = input.getInt("请输入年龄：", "年龄必须是数字，");
20         String date = input.getString("请输入生日：");
21         float salary = input.getFloat("请输入工资：", "输入的工资必须是数字，");
22         String sql = "INSERT INTO person(pid,name,age,birthday,salary) VALUES
23             (perseq.nextval,'"
24                 + name + "','" + age + "',TO_DATE('"
25                 + date + "','yyyy-mm-dd'),'"+ salary + "')";
26         System.out.println(sql);
27         // 1. 使用 Class 类加载驱动程序
28         Class.forName(DBDRIVER);
29         // 2. 连接数据库
30         // 3. Statement 接口需要通过 Connection 接口进行实例化操作
31         stmt = conn.createStatement();
32         // 执行 SQL 语句，更新数据库
33         stmt.executeUpdate(sql);
34         // 4. 关闭数据库
35         conn.close();
36     }
37 }
```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 17~25 行设置了用户输入的平台提示信息，同时接收用户输入的信息，如姓名、年龄、生日、工资等所需要的信息。

特别需要注意的是第 22~25 行，这里采用拼凑的形式进行操作，也就是说当用户将所输入的信息输入进控制台之后，创建的对象接受内容并在此拼凑成为 SQL 语句，然后通过 SQL 语句进行数据库的操作控制。

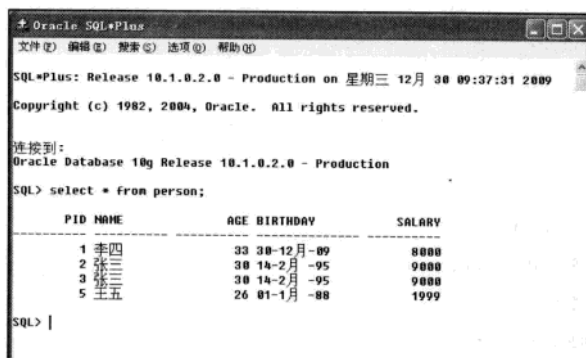
接下来通过 sqlplus 进行查询，观察输入的信息是否存入到数据库当中。

开始→运行→sqlplus。

输入用户名 system，密码 java。

进入到 SQL 控制后台，写入查询的 SQL 语句 “select * from person”。

得到的信息如图所示。



现在的程序内容是交给用户自己输入的，所以必须要考虑到用户的随意性及输入内容的不确定型。那么此时用户就可能输入一些特殊的内容。

【范例 23-9】 从控制台输入特殊内容。用户在用户名一项中输入特殊内容，比如英文名称，而且添加了单引号（代码 23-9.java）。

请输入姓名: mr'smith

请输入年龄: 10

请输入生日: 1981-09-17

请输入工资: 890

```
INSERT INTO person(pid,name,age,birthday,salary) VALUES
(perseq.nextval,'mr'smith',10,TO_DATE('1981-09-17','yyyy-mm-dd'),890.0)
```

```
Exception in thread "main" java.sql.SQLException: ORA-00917: missing comma
at
```

```
oracle.jdbc.driver.DatabaseError.throwSQLException(DatabaseError.java:124)
```

【范例分析】

因为现在是采用拼凑的 SQL 语句完成，一旦出现了单引号，拼凑出来的 SQL 语句就是一个错误的语句，所以无法执行，程序报错。

所以在实际开发中，为了避免类似的事情发生，就不会使用 Statement 进行程序编写，而是使用其子接口 PreparedStatement 进行程序的开发。

23.8 与数据库相关的接口

▶ 本节视频教学录像：12 分钟

本节介绍 PreparedStatement 接口，它是 Statement 接口的一个子接口。此接口是在实际开发中使用的最广泛的一个操作接口，采用预处理的方式完成。

23.8.1 完成增加操作

由于 Statement 存在不足，所以在今后的实际程序运行过程中，一般都采用 PreparedStatement 接口进行操作。

【范例 23-10】 运用 PreparedStatement 接口代替 Statement 接口进行增加内容操作（代码 23-10.java）。

```
01 package wf.network.prepareddemo01;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.PreparedStatement;
05 import java.util.Date;
06 public class InsertDemo {
07     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
08     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
09     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
10     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:orcl";
11     // 连接数据库的用户名
12     public static final String DBUSER = "system";
13     // 连接数据库的密码
14     public static final String DBPASS = "java";
15     public static void main(String[] args) throws Exception {
16         Connection conn = null;           // 表示数据库的连接对象
17         PreparedStatement pstmt = null;    // 表示数据库的更新操作
18         InputData input = new InputData();
19         String name = input.getString("请输入姓名：");
20         int age = input.getInt("请输入年龄：", "年龄必须是数字，");
21         Date date = input.getDate("请输入生日：", "输入的不是日期，");
```



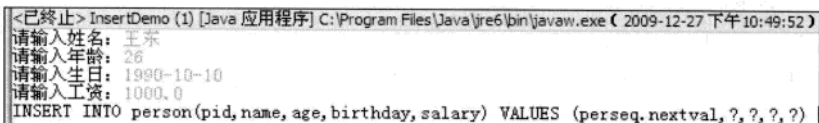
```

22     float salary = input.getFloat("请输入工资: ", "输入的工资必须是数字, ");
23     String sql = "INSERT INTO person(pid,name,age,birthday,salary) VALUES
24     (perseq.nextval,?,?,?,?) ";
25     System.out.println(sql);
26     // 1. 使用 Class 类加载驱动程序
27     Class.forName(DBDRIVER);
28     // 2. 连接数据库
29     conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
30     // 3. PreparedStatement 接口需要通过 Connection 接口进行实例化操作
31     pstmt = conn.prepareStatement(sql);           // 使用预处理的方式创建对象
32     pstmt.setString(1, name);                     // 第 1 个? 号的内容
33     pstmt.setInt(2, age);                          // 第 2 个? 号的内容
34     pstmt.setDate(3, new java.sql.Date(date.getTime()));
35     pstmt.setFloat(4, salary);
36     // 执行 SQL 语句, 更新数据库
37     pstmt.executeUpdate();
38     // 4. 关闭数据库
39     pstmt.close();
40     conn.close();
41 }
42 }

```

【运行结果】

保存并运行程序, 结果如图所示。



【代码详解】

第 17 行新建了一个 PreparedStatement 类的对象, 表示数据库的更新操作。

第 23~24 行表示 SQL 语句, 在其中的 VALUES 一项中, 表示将输入的内容传递到其中, 代码中的问号表示未知的输入项。

第 25~29 行表示将设置成功的 SQL 语句传递给数据库, 同时数据库做出相应操作。

第 31~37 行采用 PreparedStatement 类中的方法设置问号中的内容, 其中的数字 1、2、3 等表示该表每行数据的 ID 号码。

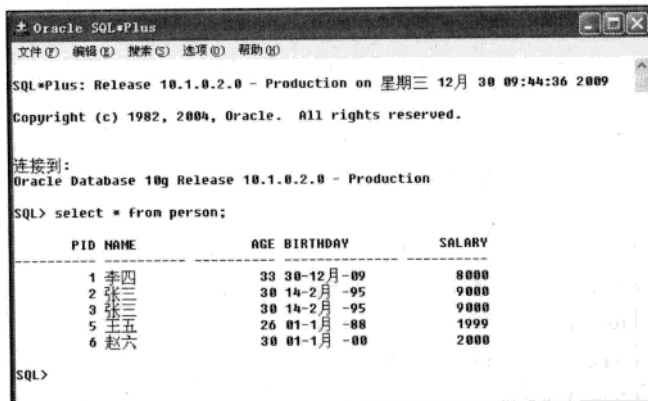
接下来通过 sqlplus 进行查询, 观察输入的信息是否存入到数据库当中。

开始→运行→sqlplus。

输入用户名 system, 密码 java。

进入到 SQL 控制后台, 写入查询的 SQL 语句 “select * from person”。

得到的信息如图所示。



23.8.2 完成查询操作

运行 PreparedStatement 接口也可以完成查询的操作, 查询的时候执行的同样是 SQL 语句, 但是如果查询全部的话, 则不需要设置任何的内容。

【范例 23-11】 运用 PreparedStatement 接口完成查询操作。运用 PreparedStatement 接口做查询操作要比用 Statement 接口直接做查询操作更有准确性和可塑性(代码 23-11.java)。

```

01 package wf.network.prepareddemo01;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.PreparedStatement;
05 import java.sql.ResultSet;
06 import java.util.Date;
07 public class SelectDemo01 {
08     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
09     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
10     // 连接地址是由各个数据库生产商单独提供的, 所以需要单独记住
11     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:ORCL";
12     // 连接数据库的用户名
13     public static final String DBUSER = "system";
14     // 连接数据库的密码
15     public static final String DBPASS = "java";
16     public static void main(String[] args) throws Exception {
17         Connection conn = null;           // 表示数据库的连接对象
18         PreparedStatement pstmt = null;    // 表示数据库的更新操作
19         ResultSet result = null;          // 表示接收数据库的查询结果
20         String sql = "SELECT pid,name,age,birthday,salary FROM person";
21         // 1. 使用 Class 类加载驱动程序

```

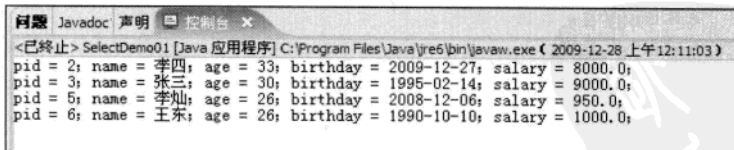
```

22     Class.forName(DBDRIVER);
23     // 2. 连接数据库
24     conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
25     // 3. PreparedStatement 接口需要通过 Connection 接口进行实例化操作
26     pstmt = conn.prepareStatement(sql);
27     // 执行 SQL 语句, 查询数据库
28     result = pstmt.executeQuery();
29     while(result.next()){                                // 是否有下一行数据
30         int pid = result.getInt(1);
31         String name = result.getString(2);
32         int age = result.getInt(3);
33         Date birthday = result.getDate(4);
34         float salary = result.getFloat(5);
35         System.out.print("pid = " + pid + "; ");
36         System.out.print("name = " + name + "; ");
37         System.out.print("age = " + age + "; ");
38         System.out.print("birthday = " + birthday + "; ");
39         System.out.println("salary = " + salary + "; ");
40     }
41     // 4. 关闭数据库
42     result.close();
43     pstmt.close();
44     conn.close();
45 }
46 }
temp

```

【运行结果】

保存并运行程序, 结果如图所示。



```

<已终止> SelectDemo01 [Java 应用程序] C:\Program Files\Java\jre6\bin\javaw.exe ( 2009-12-28 上午 12:11:03 )
pid = 2; name = 李四; age = 33; birthday = 2009-12-27; salary = 8000.0;
pid = 3; name = 张三; age = 30; birthday = 1995-02-14; salary = 9000.0;
pid = 5; name = 李灿; age = 26; birthday = 2008-12-06; salary = 950.0;
pid = 6; name = 王东; age = 26; birthday = 1990-10-10; salary = 1000.0;

```

【代码详解】

第 18 行表示新建的 PreparedStatement 类的对象。

第 26 行表示执行 SQL 语句, 进行查询数据库操作。

第 28 行将得到的查询信息储存在 result 内部, 同时运用 result 的方法进行输出打印, 将查询到的信息回馈到控制台上。

第 35~39 行为控制台输出查询到的结果。

23.8.3 完成模糊查询操作

进行模糊查询需要使用 LIKE 语句，在 LIKE 语句中则需要使用 “%” 进行匹配。

【范例 23-12】 运用 PreparedStatement 类中的方法进行模糊操作。所谓模糊，就是指有一定相似程度的值的查询，同搜索关键字进行具有关键字信息的查询类似（代码 23-12.java）。

```
01 package wf.network.prepareddemo01;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.PreparedStatement;
05 import java.sql.ResultSet;
06 import java.util.Date;
07 public class SelectDemo02 {
08     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
09     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
10     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
11     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:ORCL";
12     // 连接数据库的用户名
13     public static final String DBUSER = "system";
14     // 连接数据库的密码
15     public static final String DBPASS = "java";
16     public static void main(String[] args) throws Exception {
17         Connection conn = null;           // 表示数据库的连接对象
18         PreparedStatement pstmt = null;    // 表示数据库的更新操作
19         ResultSet result = null;           // 表示接收数据库的查询结果
20         String keyWord = "";
21         String sql = "SELECT pid,name,age,birthday,salary FROM person WHERE name
22         LIKE ? OR birthday LIKE ?";
23         // 1. 使用 Class 类加载驱动程序
24         Class.forName(DBDRIVER);
25         // 2. 连接数据库
26         conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
27         // 3. PreparedStatement 接口需要通过 Connection 接口进行实例化操作
28         pstmt = conn.prepareStatement(sql);
29         pstmt.setString(1,"%"+keyWord+"%");
30         pstmt.setString(2,"%"+keyWord+"%");
31         // 执行 SQL 语句，查询数据库
32         result = pstmt.executeQuery();
33         while(result.next()){               // 是否有下一行数据
```

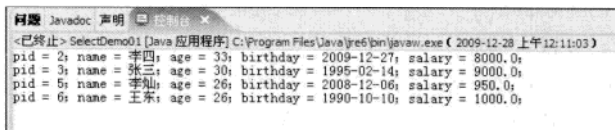
```

34      int pid = result.getInt(1);
35      String name = result.getString(2);
36      int age = result.getInt(3);
37      Date birthday = result.getDate(4);
38      float salary = result.getFloat(5);
39      System.out.print("pid = " + pid + "; ");
40      System.out.print("name = " + name + "; ");
41      System.out.print("age = " + age + "; ");
42      System.out.print("birthday = " + birthday + "; ");
43      System.out.println("salary = " + salary + "; ");
44  }
45  // 4. 关闭数据库
46  result.close();
47  pstmt.close();
48  conn.close();
49  }
50  }

```

【运行结果】

保存并运行程序，结果如图所示。



```

<已终止> SelectDemo01 [Java 应用程序] C:\Program Files\Java\jre6\bin\javaw.exe ( 2009-12-28 上午 12:11:03 )
pid = 2; name = 李四; age = 33; birthday = 2009-12-27; salary = 8000.0;
pid = 3; name = 张三; age = 30; birthday = 1995-02-14; salary = 9000.0;
pid = 5; name = 李仙; age = 26; birthday = 2008-12-06; salary = 950.0;
pid = 6; name = 王东; age = 26; birthday = 1990-10-10; salary = 1000.0;

```

【代码注解】

第 20 行表示查询的一个空值。

第 21~22 行表示通过编写 SQL 语句的 LIKE 语句进行模糊查询。由于是模糊查询，所以不知道进行哪种情况下的查询，在这里用问号表示需要查询的模糊内容。

第 29~30 行运用 preparedstatement 类中的 setString 方法进行查询操作，其中运用“%”进行匹配。

23.9 批处理

本节视频教学录像：4 分钟

批处理(Batch)也称为批处理脚本。顾名思义，批处理就是对某对象进行批量的处理。批处理文件的扩展名为 bat。目前比较常见的批处理包含两类：DOS 批处理和 PS 批处理。PS 批处理是基于强大的图片编辑软件 Photoshop 的，用来批量处理图片的脚本；而 DOS 批处理则是基于 DOS 命令的，用来自动地批量地执行 DOS 命令以实现特定操作的脚本。

在批处理中，多条 SQL 语句可以一次性执行完毕，称为批处理操作。

批处理是在 JDBC 2.0 之后提出的概念，但是在 JDBC 2.0 之中还有很多其他的内容，包括可

滚动的结果集，并使用结果集更新数据等，但是这些操作基本上都不使用。

首先在 Statement 接口上定义一个 addBatch()方法，此方法可用于加入批处理，之后使用 executeBatch()方法进行批处理的操作。

【范例 23-13】 一次性输入数据库多条语句。这是在 JDBC2.0 之后 java 提出的新概念，表示一次性可以处理多条数据库信息内容，以方便进行增加数据的操作（代码 23-13.java）。

```
01 package wf.network.prepareddemo01;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.PreparedStatement;
05 public class BatchInsertDemo {
06     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
07     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
08     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
09     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:ORCL";
10     // 连接数据库的用户名
11     public static final String DBUSER = "system";
12     // 连接数据库的密码
13     public static final String DBPASS = "java";
14     public static void main(String[] args) throws Exception {
15         Connection conn = null;           // 表示数据库的连接对象
16         PreparedStatement pstmt = null;    // 表示数据库的更新操作
17         String sql = "INSERT INTO person(pid,name,age,birthday,salary) VALUES
18             (perseq.nextval,?,?,?,?) ";
19         System.out.println(sql);
20         // 1. 使用 Class 类加载驱动程序
21         Class.forName(DBDRIVER);
22         // 2. 连接数据库
23         conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
24         // 3. PreparedStatement 接口需要通过 Connection 接口进行实例化操作
25         pstmt = conn.prepareStatement(sql); // 使用预处理的方式创建对象
26         for(int i=0;i<10;i++){
27             pstmt.setString(1, "lxh-" + i);           // 第 1 个? 号的内容
28             pstmt.setInt(2, 20 + i);                 // 第 2 个? 号的内容
29             pstmt.setDate(3, new java.sql.Date(new
30                 java.util.Date().getTime()));         // 第 3 个? 号的内容
31             pstmt.setFloat(4, 900*i);                 // 第 4 个? 号的内容
32             pstmt.addBatch();                         // 增加批处理
33         }
34         // 执行 SQL 语句，更新数据库
```



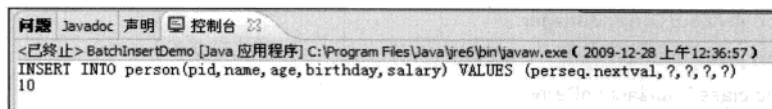
```

35     int i[] = pstmt.executeBatch();
36     System.out.println(i.length);
37     // 4. 关闭数据库
38     pstmt.close();
39     conn.close();
40 }
41 }

```

【运行结果】

保存并运行程序，结果如图所示。



【代码详解】

第 26~33 行添加了一个 for 循环，表示在数据库中循环增加数据。由于设置的 for 循环为 10 次，因此增加的数据信息也应该为 10 条。

第 27~31 行再次运用 setString() 方法设置第 18 行中 SQL 语句中的每一个问号。

第 32 行表示增加批处理。

接下来通过 sqlplus 进行查询，观察输入的信息是否存入到数据库当中。

开始→运行→sqlplus。

输入用户名 system，密码 java。

进入到 SQL 控制后台，写入查询的 SQL 语句“select * from person”。

得到的信息如图所示。

PID	NAME	AGE	BIRTHDAY	SALARY
1	李四	33	30-12月-09	9000
2	张三	30	14-2月-95	9000
3	张三	30	14-2月-95	9000
7	1kh-6	20	30-12月-09	0
5	王五	26	01-1月-08	1999
6	赵六	30	01-1月-08	2000
8	1kh-1	21	30-12月-09	900
9	1kh-2	22	30-12月-09	1000
10	1kh-3	23	30-12月-09	2700
11	1kh-4	24	30-12月-09	3600
12	1kh-5	25	30-12月-09	4500
13	1kh-6	26	30-12月-09	5400
14	1kh-7	27	30-12月-09	6300
15	1kh-8	28	30-12月-09	7200
16	1kh-9	29	30-12月-09	8100

23.10 事务处理

本节视频教学录像：7 分钟

Oracle 数据库支持事务处理，可以通过 commit 提交事务，通过 rollback 回滚事务。这两种

方式是进行事务处理的首要手段。

在 JDBC 中也同样支持事务的处理，但所有的事务处理都需要依靠 Connection 完成。否则在事务处理的过程中，会造成在错误之前的代码运行了，而错误之后的代码没有运行的情况，这在实际开发的过程当中是不允许出现的，否则会造成很大的损失。

在 Connection 操作中所有的数据库更新属于立即更新，如果想要进行事务的操作，则首先应该停止自动更新操作，之后所有的更新则通过 commit()进行提交，如果有问题则回滚。

【范例 23-14】 通过 commit () 方法提交事务（代码 23-14.java）。

```

01 package wf.network.trandemo01;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.Statement;
05 public class TransactionDemo02 {
06     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中
07     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
08     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住
09     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:ORCL";
10     // 连接数据库的用户名
11     public static final String DBUSER = "system";
12     // 连接数据库的密码
13     public static final String DBPASS = "java";
14     public static void main(String[] args) throws Exception {
15         Connection conn = null;           // 表示数据库的连接对象
16         Statement stmt = null;           // 表示数据库的更新操作
17         // 1. 使用 Class 类加载驱动程序
18         Class.forName(DBDRIVER);
19         // 2. 连接数据库
20         conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
21         conn.setAutoCommit(false);       // 取消自动提交
22         // 3. Statement 接口需要通过 Connection 接口进行实例化操作
23         stmt = conn.createStatement();
24         try{
25             stmt.addBatch("INSERT INTO person(pid,name,age,birthday,salary) VALUES
(perseq.nextval,'张三',30,TO_DATE('1995-02-14','yyyy-mm-dd'),9000.0) ");
26             stmt.addBatch("INSERT INTO person(pid,name,age,birthday,salary) VALUES
(perseq.nextval,'李四',30,TO_DATE('1995-02-14','yyyy-mm-dd'),9000.0) ");
27             stmt.addBatch("INSERT INTO person(pid,name,age,birthday,salary) VALUES
(perseq.nextval,'王五',30,TO_DATE('1995-02-14','yyyy-mm-dd'),9000.0) ");
28             stmt.addBatch("INSERT INTO person(pid,name,age,birthday,salary) VALUES
(perseq.nextval,'赵六',30,TO_DATE('1995-02-14','yyyy-mm-dd'),9000.0) ");

```

```

29          stmt.addBatch("INSERT INTO person(pid,name,age,birthday,salary) VALUES
(perseq.nextval,'孙七',30,TO_DATE('1995-02-14','yyyy-mm-dd'),9000.0)");
30      // 执行 SQL 语句, 更新数据库
31      int i[] = stmt.executeBatch();
32      System.out.println(i.length);
33      conn.commit();           // 提交
34  }catch(Exception e){
35      conn.rollback();        // 回滚
36  }
37  // 4. 关闭数据库
38  stmt.close();
39  conn.close();
40  }
41  }

```

【代码详解】

第 21 行进行取消自动更新处理。否则在实际操作的过程中, 如果遇见错误则可能造成在错误之前的语句执行了, 而错误之后未执行的情况。

第 24~44 行对提交的 SQL 语句进行试验操作, 如果在提交的 SQL 语句中出现错误, 则在第 45 行中进行回滚, 此时 SQL 语句不会进行提交。如果没有出现错误, 则在第 43 行中进行提交。

23.11 MySQL 数据库

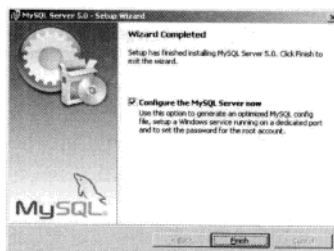
▶ 本节视频教学录像: 20 分钟

Oracle 数据库本身是一个非常大的数据库, 而对于一些小型的开发, 使用此数据库肯定不合适, 所以在实际中使用 mysql 的特别多。

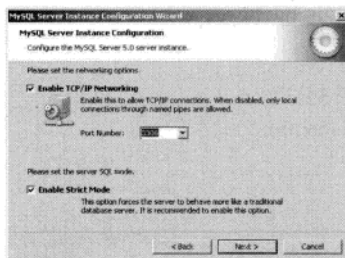
MySQL 数据库本身是免费的, 而且最新的版本的功能也足够强大, 所以掌握 MYSQL 是很有必要的。

现在的 MySQL 已经被 SUN 收购。

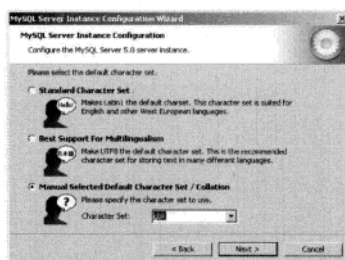
23.11.1 MySQL 数据库的安装



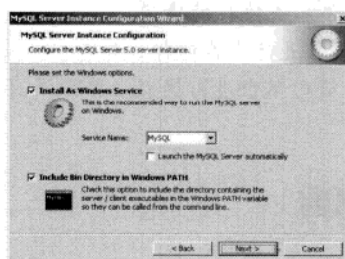
MySQL 安装的过程非常简单，但是需要进行合理的配置。



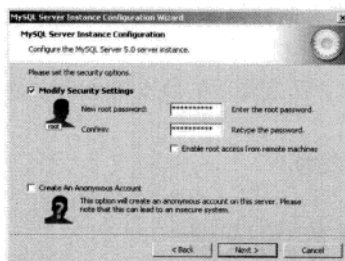
MySQL 数据库的默认端口号是 3306。



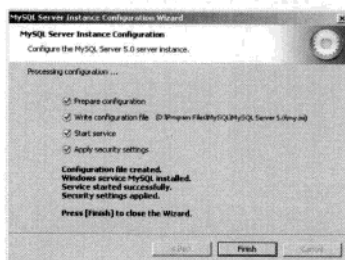
选择数据库的编码为 gbk，否则程序在读取的时候会出现乱码问题。



上图中的两个复选框均勾选。



设置 mysql 中 root 用户的密码为 java。



23.11.2 MySQL 数据库的基本命令

在 MySQL 的基本操作命令中除了一些特殊的之外，其他的都是标准的 SQL 语句。

单击【开始】按钮，选择【运行】命令，弹出【运行】对话框，在【打开】文本框中输入“cmd”，然后单击【确定】按钮。接下来进行如下操作。

1. 连接数据库

```
mysql -u 用户名 -p 密码
```

在 cmd 命令行中输入：

```
mysql -uroot -pjava
```

出现进入数据库提示。

2. 查看全部的数据库

```
show databases ;
```

3. 创建数据库

```
CREATE DATABASE orcl ;
```

4. 使用数据库

```
USE orcl ;
```

5. 查看全部的表

```
SHOW TABLES ;
```

6. 创建一张表，其中 id 可以自动增长

```
DROP TABLE person ; -- 删除表
```

```
CREATE TABLE person(
```

```
    pid      INT      AUTO_INCREMENT  PRIMARY KEY NOT NULL ,
```

```
    name     VARCHAR(50) NOT NULL ,
```

```
    age      INT      NOT NULL ,
```

```
    birthday DATE     NOT NULL ,
```

```
    salary   FLOAT    NOT NULL
```

```
);
```

7. 查看数据表结构

```
DESC 表名称 ;
```


8. 插入数据

```
INSERT INTO person(name,age,birthday,salary) VALUES ('张三',30,'1992-02-24',9000.0) ;
```

23.11.3 使用 MySQL 数据库

直接配置驱动程序，但 MySQL 驱动程序需要从网上单独下载。

【范例 23-15】 运用 java 连接 MySQL 数据库（代码 23-15.java）。

```
01 package org.lxh.mysqlDemo;  
02 import java.sql.Connection;  
03 import java.sql.DriverManager;  
04 import java.sql.PreparedStatement;  
05 import java.text.SimpleDateFormat;  
06 import java.util.Date;  
07 public class JDBCMySQL {  
08     // 驱动程序就是之前在 classpath 中配置的 jdbc 的驱动程序的 jar 包中  
09     public static final String DBDRIVER = "org.gjt.mm.mysql.Driver";  
10     // 连接地址是由各个数据库生产商单独提供的，所以需要单独记住  
11     public static final String DBURL = "jdbc:mysql://localhost:3306/orcl";  
12     // 连接数据库的用户名  
13     public static final String DBUSER = "root";  
14     // 连接数据库的密码  
15     public static final String DBPASS = "java";  
16     public static void main(String[] args) throws Exception {  
17         Connection conn = null;           // 表示数据库的连接对象  
18         PreparedStatement pstmt = null;    // 表示数据库的更新操作  
19         String name = "张三";  
20         int age = 30;  
21         Date date = new SimpleDateFormat("yyyy-MM-dd").parse("1983-02-15");  
22         float salary = 7000.0f;  
23         String sql = "INSERT INTO person(name,age,birthday,salary) VALUES  
24             (?, ?, ?, ?)";  
25         System.out.println(sql);  
26         // 1. 使用 Class 类加载驱动程序  
27         Class.forName(DBDRIVER);  
28         // 2. 连接数据库  
29         conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);  
30         // 3. PreparedStatement 接口需要通过 Connection 接口进行实例化操作  
31         pstmt = conn.prepareStatement(sql);    // 使用预处理的方式创建对象
```

```
32      pstmt.setString(1, name);           // 第 1 个? 号的内容
33      pstmt.setInt(2, age);                // 第 2 个? 号的内容
34      pstmt.setDate(3, new java.sql.Date(date.getTime()));
35      pstmt.setFloat(4, salary);
36      // 执行 SQL 语句, 更新数据库
37      pstmt.executeUpdate();
38      // 4. 关闭数据库
39      pstmt.close();
40      conn.close();
41  }
42 }
```

【代码详解】

和连接 Oracle 数据库相同。

第 9~10 行输入数据库的驱动程序地址及数据库连接信息。

第 13~15 行输入连接数据库的用户名 root 和密码 java。

23.12 练一练

一、填空题

1. Oracle 数据库的端口号为_____。
2. _____方法可以进行数据库的更新操作。
3. PreparedStatement 类为_____类的子类。
4. 增加新数据的 SQL 语句命令是_____。

二、简答题

1. 简述 Oracle 的端口号及地址信息。
2. PreparedStatement 与 Statement 类有什么不同?
3. 在事务处理过程中, 为什么要通过 setAutocommit 方法关闭自动更新? 这样做有什么好处?

23.13 跟我上机

尝试安装 Oracle 数据库及 MySQL 数据库, 并通过 Java 进行连接。



读书笔记

第 4 篇

项目实战

在本篇中,将综合前面所学的各种基础知识以及高级开发技巧来实际开发应用程序。通过本篇的学习,读者将对 Java 中的软件开发拥有切身的体会,并为日后进行项目开发积累下实战经验。

- ▶ 第 24 章 Java 项目开发实战——五子棋游戏
- ▶ 第 25 章 Java 项目开发实战——人事管理

第 24 章

Java项目开发实战——五子棋游戏



本章视频教学录像：4 小时 39 分钟

师傅领进门，修行在个人。至此，关于Java的基础知识和各项专题都已经进行了系统的学习。希望前面所讲的知识能起到抛砖引玉的作用。读者可以根据以前学到的经验融会贯通，不断攻克新的技术。本书在前面未提及用户界面设计的技术——Swing，但通过以前的知识和经验的积累，本章带领大家进行一个开发实战，在实例中学习这些技术。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握 Swing 编程的相关知识
- ☐ 掌握 Java 中显示图片类 ImageIO 的使用
- ☐ 掌握 Java 中绘制图片类 Graphics 的使用
- ☐ 掌握游戏界面的开发
- ☐ 掌握开发过程中出现问题的解决方式



24.1 系统概述

本节视频教学录像：39 分钟

本系统是一个教学案例。读者可以先试运行本书提供的源码，以对需要开发的程序的功能有一个大致的了解。

24.1.1 运行本系统

首先大家要学会如何运行本系统，可对本程序的功能有所了解。下面简述运行的具体步骤。

- ① 把光盘中的“final\ch24\FiveChessProject”目录复制到硬盘中，本例使用“E:\”。
- ② 运行 Eclipse，新建一个 Java 项目，本例命名为“WuZi”（注意：尽量不要命名为 FiveChessProject，与本书开发的系统重名，否则有可能造成以后不能新建以这个名字命名的项目）。勾选【从现有资源创建项目】单选按钮，在【目录】文本框中输入“E:\FiveChessProject”，然后单击【完成】按钮。
- ③ 在【包资源管理器】中选择【WuZi】>【org.liky.game.test】>【Test.java】，右击该项，然后在弹出的快捷菜单中选择【运行方式】>【1 Java 应用程序】菜单项，即可出现五子棋游戏界面。
- ④ 后面就可以玩一玩该游戏，测试一下它的功能了。

24.1.2 本系统的开发步骤

开发本程序的主要步骤如下。

- ① 新建项目，并新建一个测试类用于研究技术和创建五子棋窗体。
- ② 新建一个类 FiveChessFrame，大部分程序代码都要在这个类中完成。
- ③ 测试代码。
- ④ 运行程序。

下面给出第①和第②步的详细步骤。

- ① 建立新的 Java 项目“FiveChessProject”。

这里的工作空间指定为“D:\java\ch24\final”。

- ② 在该项目中创建一个供测试的类“Test”，该类负责实例化五子棋主窗口。另外在研究 Swing 等编程技术时也会用到该类来运行一些测试代码。

其参数如下。

包：org.liky.game.test

名称: Test

复选 “public static void main(String[] args)” 项。

其他选项为默认设置即可。

- ③ 在该项目中创建一个类 “FiveChessFrame”，该类负责五子棋主窗口的显示和控制，是主程序，是我们学习的重点。在后面会详细地介绍这里的代码设计。

其参数如下。

包: org.liky.game.frame

名称: FiveChessFrame

其他选项为默认设置即可。

24.1.3 五子棋游戏的功能

五子棋游戏的基本功能如下。

- (1) 在单击鼠标时，在相应的位置显示棋子。
- (2) 自动判断游戏是否结束，是否黑方或白方已经胜利。
- (3) 对游戏时间进行设置，判断是否超出规定的时间。

24.1.4 主要技术

本程序主要会用到以下 3 种技术。

- (1) Swing 编程。
- (2) ImageIO 类的使用。
- (3) 图片的绘制。

24.2 开发前的知识准备之一——Swing 编程

本节视频教学录像：43 分钟

Swing 是一个用于开发 Java 应用程序用户界面的开发工具包。它以抽象窗口工具包(AWT)为基础，使跨平台应用程序可以使用统一的外观风格。Swing 开发人员只需用很少的代码，就可以利用 Swing 丰富、灵活的功能和模块化组件来创建优雅的用户界面。

图形用户接口(GUI)库最初的设计目的是让程序员构建一个通用的 GUI，使其在所有的平台上都能够正常显示。但是比较遗憾的是，AWT 产生的是在各系统看来都同样欠佳的图形用户接口，Java 1.2 为老的 Java 1.0 AWT 添加了 Java 基础类(JFC)，这是一个被称为 “Swing” 的 GUI 的一部分。Swing 是第二代 GUI 开发工具集，AWT 采用了与特定平台相关的实现，而绝大部分 Swing 组件却不是。Swing 是构筑在 AWT 上层的一组 GUI 组件的集合，为了保证其可移植性，它完全用 Java 语言编写，与 AWT 相比，Swing 提供了更完整的组件，引入了许多新的特性和能力。Swing 提供了更多的组件库，如 JTable、JTree、JCombox 等。Swing 也增强了 AWT 中组件的功能。正因为 Swing 具备了如此多的优势，所以我们以后在开发中都使用 Swing。JComponent 类是 Swing 组件的基类，而 JComponent 类继承自 Container 类，因此所有的 Swing 组件都是 AWT

的容器。

24.2.1 与窗体相关的类——JFrame

JFrame 是创建窗体的 swing 类，存在于 javax.swing.JFrame 包中。用来创建一个图形界面的原始窗口，并设置其大小和位置等属性，是 Swing 编程的基础类之一。

1. JFrame 中的主要方法如下。

setVisible(): 设置窗体是否显示。
 setTitle(): 设置窗体标题。
 setSize(): 设置窗体大小。
 setLocation(): 设置窗体初始显示的位置。
 setResizable(): 设置窗体是否可以改变大小。
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE): 设置窗体关闭方式，关闭窗体时同时结束程序。

2. 和 JFrame 相关的取得屏幕大小的静态方法如下。

Toolkit.getDefaultToolkit().getScreenSize().width: 取得当前屏幕的宽度。
 Toolkit.getDefaultToolkit().getScreenSize().height: 取得当前屏幕的高度。

下面给出一些范例来验证以上提到的方法。上接 24.1.2 小节的第②步，已经建立了测试类。

【范例 24-1】 生成一个窗体并显示到屏幕上。在测试类文件 Test.java 中输入以下代码（代码 24-1.java）。

```
01 package org.liky.game;
02 import javax.swing.JFrame;
03 public class Test {
04     public static void main(String[] args) {
05         // TODO 自动生成方法存根
06         JFrame jf = new JFrame();
07         jf.setVisible(true);
08     }
09 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

- (1) 使用第 2 句代码即可生成一个窗体，该窗体出现在屏幕的左上角。
- (2) 只有使用了 setVisible 方法并且传入 true 参数后，窗体才能被显示出来。

(3) 不要忘记添加 JFrame 的引用。

```
import javax.swing.JFrame;
```

【范例 24-2】 设置窗体的标题、位置和大小。在测试类文件 Test.java 中输入以下代码（代码 24-2.java）。

```
01 package org.liky.game;
02 import javax.swing.JFrame;
03 public class Test {
04     public static void main(String[] args) {
05         JFrame jf = new JFrame();
06         jf.setVisible(true);
07         jf.setTitle("五子棋");
08         jf.setSize(200,100);
09         jf.setLocation(200,100);
10     }
11 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

- (1) setTitle 方法可用来设置窗体的标题，直接输入所需的字符串即可。
- (2) setLocation 方法可用来确定窗体的左上角的坐标，以像素为单位，这样就确定了窗体的显示位置。
- (3) SetSize 方法可用来确定窗体的长度和宽度，以像素为单位，这样就确定了窗体的大小。

【范例 24-3】 使窗体不可调整大小，并且在关闭窗口时关闭应用程序。在测试类文件 Test.java 中输入以下代码（代码 24-3.java）。

```
01 package org.liky.game;
02 import javax.swing.JFrame;
03 public class Test {
04     public static void main(String[] args) {
05         JFrame jf = new JFrame();
06         jf.setVisible(true);
07         jf.setTitle("五子棋");
```



```

08      jf.setSize(200,100);
09      jf.setLocation(200,100);
10      jf.setResizable(false);
11      jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12  }
13  }

```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

(1) `jf.setResizable(false)`这句代码的意思是使窗体不能由用户来改变大小，以免造成以后的五子棋棋盘界面混乱。

(2) `jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`这句代码的意思是在关闭窗体的同时，关闭应用程序以节省资源。

虽然现在已经能把窗口定位，但却不能把窗体放在屏幕的正中央。原因是用户的屏幕分辨率不一定是相同的，照顾到一种分辨率，必然就照顾不到第2种分辨率。这就要求程序能自动检测当前屏幕的分辨率，并根据检测结果进行计算，把窗体放在屏幕的正中央。

【范例 24-4】 得到当前屏幕的分辨率。在测试类文件 `Test.java` 中输入以下代码(代码 24-4.java)。

```

01  package org.liky.game;
02  import java.awt.Toolkit;
03  import javax.swing.JFrame;
04  public class Test {
05      public static void main(String[] args) {
06          JFrame jf = new JFrame();
07          jf.setVisible(true);
08          jf.setTitle("五子棋");
09          jf.setSize(200,100);
10          jf.setLocation(200,100);
11          jf.setResizable(false);
12          jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13          int width = Toolkit.getDefaultToolkit().getScreenSize().width;
14          int height = Toolkit.getDefaultToolkit().getScreenSize().height;
15          System.out.println("宽度为: "+ width);
16          System.out.println("高度为: "+ height);

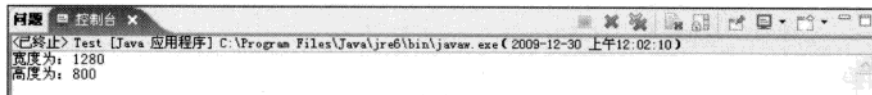
```



```
17     }
18 }
```

【运行结果】

保存并运行程序，结果如图所示。



【范例分析】

- (1) 本范例运行结果是在控制台上的结果，程序的界面和上一个范例的相同，这里就不再提及。
 - (2) 控制台窗口中输出了我们关心的问题：当前屏幕的分辨率。
 - (3) Toolkit.getDefaultToolkit().getScreenSize()是一个静态方法，无需实例化即可使用该方法，用于得到当前屏幕的分辨率。
- 有了屏幕分辨率，把窗口显示在屏幕中央只是一个算法的问题。

【范例 24-5】 把窗口放在屏幕的正中央。在测试类文件 Test.Java 中输入以下代码（代码 24-5.java）。

```
01 package org.liky.game;
02 import java.awt.Toolkit;
03 import javax.swing.JFrame;
04 public class Test {
05     public static void main(String[] args) {
06         JFrame jf = new JFrame();
07         jf.setVisible(true);
08         jf.setTitle("五子棋");
09         jf.setSize(200,100);
10         //jf.setLocation(200,100);
11         jf.setResizable(false);
12         jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         int width = Toolkit.getDefaultToolkit().getScreenSize().width;
14         int height = Toolkit.getDefaultToolkit().getScreenSize().height;
15         System.out.println("宽度为: "+ width);
16         System.out.println("高度为: "+ height);
17         jf.setLocation((width-200)/2,(height-100)/2);
18     }
19 }
```

【运行结果】

保存并运行程序，结果就是窗口确实在屏幕的正中央。

【范例分析】

(1) (屏幕宽度-窗体宽度)/2 即可得到窗体位于屏幕中央位置的横坐标, 把这个参数传给 setLocation 方法。

(2) (屏幕高度-窗体高度)/2 即可得到窗体位于屏幕中央位置的纵坐标, 把这个参数传给 setLocation 方法。

至此关于 Swing 界面方面的准备知识介绍完毕。但现在还有一个问题: 程序的主窗体在主程序的 main 方法中建立显然不合适, 不符合面向对象的编程思想。为此需要把主窗体包装成类, 封装起来。



注意: 这个步骤大家一定要做, 以后的程序开发都是以这个操作为基础的。

【范例 24-6】 把主窗体封装成一个类。

① 在测试类文件 Test.java 中输入以下代码 (代码 24-6.java)。

```
01 package org.liky.game.test;
02 import org.liky.game.frame.FiveChessFrame;
03 public class Test {
04     public static void main(String[] args) {
05         FiveChessFrame ff = new FiveChessFrame();
06     }
07 }
```

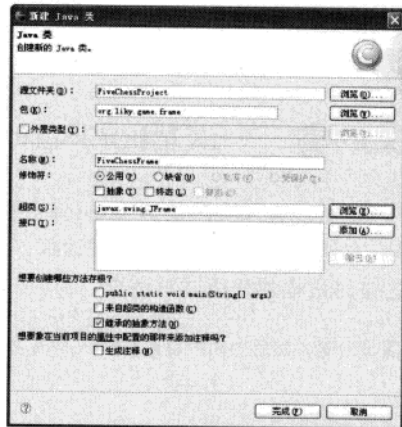
② 新建一个类 FiveChessFrame。

其参数如下。

包: org.liky.game.frame

名称: FiveChessFrame

其他选项为默认设置即可。



③ 在该类文件中输入如下代码（代码 24-7.java）。

```
01 package org.liky.game.frame;
02 import java.awt.Toolkit;
03 import javax.swing.JFrame;
04 public class FiveChessFrame extends JFrame {
05     // 取得屏幕的宽度
06     int width = Toolkit.getDefaultToolkit().getScreenSize().width;
07     // 取得屏幕的高度
08     int height = Toolkit.getDefaultToolkit().getScreenSize().height;
09     public FiveChessFrame() {
10         // 设置标题
11         this.setTitle("五子棋");
12         // 设置窗体大小
13         this.setSize(500, 500);
14         // 设置窗体出现位置
15         this.setLocation((width - 500) / 2, (height - 500) / 2);
16         // 将窗体设置为大小不可改变
17         this.setResizable(false);
18         // 将窗体的关闭方式设置为默认关闭后程序结束
19         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20         // 将窗体显示出来
21         this.setVisible(true);
22     }
23 }
```

【运行结果】

该程序的运行结果和【范例 24-5】的结果一致。

【范例分析】

把程序主窗体包装成类，以后我们只需要在 FiveChessFrame.java 文件中完善代码即可。

24.2.2 与对话框相关的类——JOptionPane

在 Swing 编程中提供了 JOptionPane 类来实现类似 Windows 平台下的 MessageBox 的功能。利用 JOptionPane 类中的各个静态方法来生成各种标准的对话框，实现显示出信息、提出问题、警告和用户输入参数等功能。这些对话框都是模式对话框。

ConfirmDialog：确认对话框，提出问题，然后由用户自己来确认（按“Yes”或“No”按钮）。

InputDialog：提示输入文本。

MessageDialog：显示信息。

OptionDialog: 组合其他 3 个对话框类型。

这 4 个对话框可以采用 showXXXDialog() 来显示, 如 showConfirmDialog() 显示确认对话框, showInputDialog() 显示输入文本对话框, showMessageDialog() 显示信息对话框, showOptionDialog() 显示选择性的对话框。

它们所使用的参数说明如下。

(1) **ParentComponent**: 指示对话框的父窗口对象, 一般为当前窗口。也可以为 null, 即采用默认的 Frame 作为父窗口, 此时对话框将设置在屏幕的正中。

(2) **message**: 指示要在对话框内显示的描述性的文字。

(3) **String title**: 标题条文字串。

(4) **Component**: 在对话框内要显示的组件 (如按钮)。

(5) **Icon**: 在对话框内要显示的图标。

(6) **messageType**: 一般可以为 ERROR_MESSAGE、INFORMATION_MESSAGE、WARNING_MESSAGE、QUESTION_MESSAGE 和 PLAIN_MESSAGE 等值。

(7) **optionType**: 它决定在对话框的底部所要显示的按钮选项。一般可以为 DEFAULT_OPTION、YES_NO_OPTION、YES_NO_CANCEL_OPTION、OK_CANCEL_OPTION 等。

【范例 24-7】 测试由 JOptionPane 产生的各种对话框。在测试类文件 Test.java 中输入以下代码 (代码 24-8.java)。

```
01 package org.liky.game;
02 import java.awt.Toolkit;
03 import javax.swing.JFrame;
04 import javax.swing.JOptionPane;
05 import org.liky.game.frame.FiveChessFrame;
06 public class Test {
07     public static void main(String[] args) {
08         FiveChessFrame ff = new FiveChessFrame();
09         JOptionPane.showMessageDialog(ff, "我的信息");
10         int result=JOptionPane.showConfirmDialog(ff,"我的确认信息：现在要开始游戏吗？");
11         if (result == 0){
12             JOptionPane.showMessageDialog(ff, "游戏开始");
13         }
14         if (result == 1){
15             JOptionPane.showMessageDialog(ff, "游戏结束");
16         }
17         if (result == 2){
18             JOptionPane.showMessageDialog(ff, "请重新选择");
19         }
20         String username = JOptionPane.showInputDialog("请输入你的姓名：");
21         if (username != null ){
```

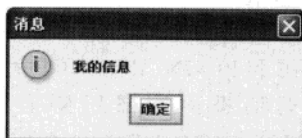


```

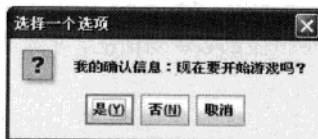
22         System.out.println(username);
23         JOptionPane.showMessageDialog(ff,"输入的姓名为: " + username );
24     }else {
25         JOptionPane.showMessageDialog(ff, "请重新输入你的姓名!");
26     }
27 }
28 }
    
```

【运行结果】

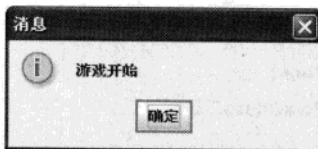
showMessageDialog(ff, " 我的信息 ") : 显示一个【消息】对话框，主要用来提示信息。



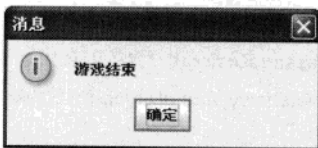
单击【确定】按钮，将会出现【选择一个选项】对话框。



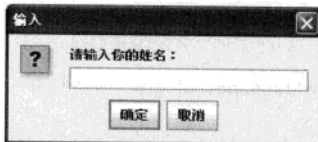
若单击【是】按钮，将会出现【游戏开始】对话框。



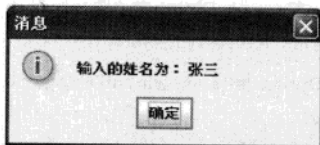
若单击【否】按钮，将会出现【游戏结束】对话框。



单击【确定】按钮，将会出现【请输入你的姓名:】对话框。



输入“张三”，将会出现【输入的姓名为: 张三】对话框。



【范例分析】

`showConfirmDialog(mf, " 我的确认信息,现在要开始游戏吗? ");`

显示一个确认对话框, 用户选择后, 根据返回的结果进行判断。

`showInputDialog("请输入你的姓名: ");`

显示一个信息输入对话框, 作用是用来保存用户输入的信息。

24.2.3 与监听鼠标相关的类——MouseListener

MouseListener 是一个接口, 用于接收组件上“感兴趣”的鼠标事件(按下、释放、单击、进入或离开)的侦听器接口。

旨在处理鼠标事件的类要么实现此接口(及其包含的所有方法), 要么扩展抽象类 MouseAdapter (仅重写所需的方法)。

然后使用组件的 `addMouseListener` 方法将从该类所创建的侦听器对象向该组件注册。当按下、释放或单击(按下并释放)鼠标时会生成鼠标事件, 鼠标光标进入或离开组件时也会生成鼠标事件。发生鼠标事件时, 将调用该侦听器对象中的相应方法, 并将 `MouseEvent` 传递给该方法。

要想使用 MouseListener, 需要调用 JFrame 的 `addMouseListener` 方法加入监听。加入监听器之后, 必须要实现以下 5 个方法。

`mouseClicked(MouseEvent e)` : 监听鼠标单击事件的操作。

`mouseEntered(MouseEvent e)` : 监听鼠标进入事件的操作。

`mouseExited(MouseEvent e)` : 监听鼠标离开事件的操作。

`mousePressed(MouseEvent e)` : 监听鼠标按下事件的操作。

`mouseReleased(MouseEvent e)`: 监听鼠标抬起时间的操作。

鼠标单击时执行的顺序如下。

`mousePressed > mouseReleased > mouseClicked` (判断按下与抬起是否在同一位置)

下面来看一个例子。

【范例 24-8】 测试 MouseListener。在主窗口类文件 `FiveChessFrame.java` 中输入以下代码(代码 24-9.java)。

需要注意的是: 测试 MouseListener 时, 就不能在 `Test.java` 中书写代码了。要在【范例 24-6】之后, 把主窗体封装成类后, 在 `FiveChessFrame.java` 中书写代码。

```
01 package org.liky.game.frame;
02 import java.awt.HeadlessException;
03 import java.awt.Toolkit;
04 import java.awt.event.MouseEvent;
05 import java.awt.event.MouseListener;
06 import javax.swing.JFrame;
```

```
07 import javax.swing.JOptionPane;
08 public class FiveChessFrame extends JFrame implements MouseListener {
09     // 取得屏幕的宽度
10     int width = Toolkit.getDefaultToolkit().getScreenSize().width;
11     // 取得屏幕的高度
12     int height = Toolkit.getDefaultToolkit().getScreenSize().height;
13     // 保存棋子的坐标
14     int x = 0;
15     int y = 0;
16     // 保存之前下过的全部棋子的坐标
17     // 其中数据内容 0: 表示这个点并没有棋子, 1: 表示这个点是黑子, 2: 表示这个点是白子
18     int[][] allChess = new int[19][19];
19     // 标识当前应该黑棋还是白棋下下一步
20     boolean isBlack = true;
21     // 标识当前游戏是否可以继续
22     boolean canPlay = true;
23     // 保存显示的提示信息
24     String message = "黑方先行";
25     // 保存最多拥有多少时间(秒)
26     int maxTime = 0;
27     // 保存黑方与白方的剩余时间
28     int blackTime = 0;
29     int whiteTime = 0;
30     // 保存双方剩余时间的显示信息
31     String blackMessage = "无限制";
32     String whiteMessage = "无限制";
33     public FiveChessFrame() throws HeadlessException {
34         // 设置标题
35         this.setTitle("五子棋");
36         // 设置窗体大小
37         this.setSize(500, 500);
38         // 设置窗体出现位置
39         this.setLocation((width - 500) / 2, (height - 500) / 2);
40         // 将窗体设置为大小不可改变
41         this.setResizable(false);
42         // 将窗体的关闭方式设置为默认关闭后程序结束
43         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
44         // 为窗体加入监听器
45         this.addMouseListener(this);
46         // 将窗体显示出来
47         this.setVisible(true);
```

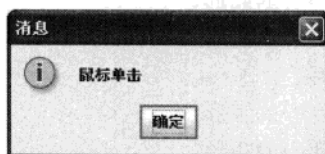
```

48     }
49     public void mouseClicked(MouseEvent arg0) {
50         // TODO 自动生成方法存根
51         JOptionPane.showMessageDialog(this,"鼠标单击");
52     }
53     public void mouseEntered(MouseEvent arg0) {
54         // TODO 自动生成方法存根
55     }
56     public void mouseExited(MouseEvent arg0) {
57         // TODO 自动生成方法存根
58     }
59     public void mousePressed(MouseEvent arg0) {
60         // TODO 自动生成方法存根
61     }
62     public void mouseReleased(MouseEvent arg0) {
63         // TODO 自动生成方法存根
64     }
65 }

```

【运行结果】

当在主窗体中单击的时候，就会出现【消息】对话框。



24.2.4 确定鼠标坐标的类——MouseEvent

`getX()` : 得到鼠标的横向位置坐标。

`getY()` : 得到鼠标的纵向位置坐标。

大家可能已经注意到，在【范例 24-8】中实现 `public void mouseEntered(MouseEvent arg0)` 方法时传递了一个参数。

`MouseEvent arg0` 这个参数可以得到鼠标的坐标。

`arg0.getX()` 可以得到鼠标的横坐标。

`arg0.getY()` 可以得到鼠标的纵坐标。



注意：由于 JDK 的版本不同，因此 `MouseEvent arg0` 可能被写成其他的形式，比如 `MouseEvent e`。不过不管是什么，总是能得到我们想要的值。

24.3 开发前的知识准备之二——显示图片的类 ImageIO

ImageIO 类包含一些用来查找 ImageReader 和 ImageWriter 以及执行简单编码和解码的静态便捷方法。

将硬盘中的图片导入到窗口中：通过 ImageIO 输入一个 BufferedImage。

drawImage()方法：将一个 BufferedImage 对象绘制到窗口中。

其他会用到的方法如下。

setColor(): 设置颜色。

setFont(): 设置字体。

24.4 开发前的知识准备之三——图形的绘制类 Graphics

图形环境的概念同在 GUI 平台上开发应用程序紧密相关。虽然通常将窗口和组件本身作为对象来表达，但仍然需要另一个接口来进行实际的绘制、着色以及文本输出等操作。Java 语言中提供这些功能的基类称做 java.awt.Graphics。从 java.awt.Component 类（所有窗口对象的基类）继承的类提供了一个名为 paint() 的方法，在需要重新绘制组件时，调用该方法。

paint() 方法只有一个参数，该参数是 Graphics 类的实例。下面介绍几个程序用到的方法。

drawString(): 绘制字符串。

drawOval(): 绘制一个空心的圆形。

fillOval(): 绘制一个实心的圆形。

drawLine(): 绘制一条线。

drawRect(): 绘制一个空心的矩形。

fillRect(): 绘制一个实心的矩形。

drawImage(): 绘制一个已经存在的图片，将一个图片直接显示到窗体中。

setColor(): 设置画笔的颜色。

setFont(): 设置绘制文字的字体。

24.5 游戏界面开发

本节视频教学录像：40 分钟

首先开发出游戏界面。

计算棋盘上每一条线的间距：这里用的是 19×19 的围棋棋盘。

总宽度为 360 像素，分成 18 份，每份是 20 像素。

总高度同样为 360 像素，分成 18 份，每份是 20 像素。

代码如下（以后将不给出代码片段，读者可以在结果文件中直接复制代码）。

```
01 public void paint(Graphics g) {  
02     // 双缓冲技术防止屏幕闪烁
```



```

03      BufferedImage bi = new BufferedImage(500, 500,
04          BufferedImage.TYPE_INT_RGB);
05      Graphics g2 = bi.createGraphics();
06      g2.setColor(Color.BLACK);
07      // 绘制背景
08      g2.drawImage(bgImage, 1, 20, this);
09      // 输出标题信息
10      g2.setFont(new Font("黑体", Font.BOLD, 20));
11      g2.drawString("游戏信息: " + message, 130, 60);
12      // 输出时间信息
13      g2.setFont(new Font("宋体", 0, 14));
14      g2.drawString("黑方时间: " + blackMessage, 30, 470);
15      g2.drawString("白方时间: " + whiteMessage, 260, 470);
16      // 绘制棋盘
17      for (int i = 0; i < 19; i++) {
18          g2.drawLine(10, 70 + 20 * i, 370, 70 + 20 * i);
19          g2.drawLine(10 + 20 * i, 70, 10 + 20 * i, 430);
20      }
21      // 标注点位
22      g2.fillOval(68, 128, 4, 4);
23      g2.fillOval(308, 128, 4, 4);
24      g2.fillOval(308, 368, 4, 4);
25      g2.fillOval(68, 368, 4, 4);
26      g2.fillOval(308, 248, 4, 4);
27      g2.fillOval(188, 128, 4, 4);
28      g2.fillOval(68, 248, 4, 4);
29      g2.fillOval(188, 368, 4, 4);
30      g2.fillOval(188, 248, 4, 4);
31      /*
32       * //绘制棋子 x = (x - 10) / 20 * 20 + 10 ; y = (y - 70) / 20 * 20 + 70 ;
33       * //黑子 g.fillOval(x - 7, y - 7, 14, 14); //白子 g.setColor(Color.WHITE);
34       * g.fillOval(x - 7, y - 7, 14, 14); g.setColor(Color.BLACK);
35       * g.drawOval(x - 7, y - 7, 14, 14);
36       */
37      // 绘制全部棋子
38      for (int i = 0; i < 19; i++) {
39          for (int j = 0; j < 19; j++) {
40              if (allChess[i][j] == 1) {
41                  // 黑子
42                  int tempX = i * 20 + 10;

```



```

43             int tempY = j * 20 + 70;
44             g2.fillOval(tempX - 7, tempY - 7, 14, 14);
45         }
46         if (allChess[i][j] == 2) {
47             // 白子
48             int tempX = i * 20 + 10;
49             int tempY = j * 20 + 70;
50             g2.setColor(Color.WHITE);
51             g2.fillOval(tempX - 7, tempY - 7, 14, 14);
52             g2.setColor(Color.BLACK);
53             g2.drawOval(tempX - 7, tempY - 7, 14, 14);
54         }
55     }
56 }
57 g.drawImage(bi, 0, 0, this);
58 }

```

24.6 绘制棋子

▶ 本节视频教学录像：13 分钟

在棋盘上的鼠标单击位置显示一个棋子。这需要在 `public void mousePressed(MouseEvent e)` 事件中书写代码。`mousePressed` 代表鼠标按下操作，当鼠标被按下时将会绘制一个棋子。具体绘制棋子的详细代码参见 24.11.5 小节。

其中，黑子用一个实心的黑圆来表示，白子用一个空心的黑圆加一个实心的白圆来表示。

注意代码中用到了 `repaint()` 方法，表示重新执行一次 `paint()` 方法。例如：`this.repaint()`；就代表立刻执行 `paint()` 方法中的代码。

24.7 保存棋局

▶ 本节视频教学录像：22 分钟

保存之前下过的棋子。由于棋子都有 x 和 y 坐标，所以要通过一个二维的数组来保存之前下过的所有棋子。数据原型为 `int[][] allChess = new int[19][19]`。19 代表棋盘的大小，数组中各个元素的值，约定 0 表示这个点并没有棋子，1 表示这个点是黑子，2 表示这个点是白子。

24.8 判断游戏胜负

▶ 本节视频教学录像：23 分钟

依据五子棋的基本游戏规则，判断是否有同一颜色的棋子连成 5 个，这是五子棋游戏的核心算法。

当游戏者下了一步棋之后，判断游戏胜负至少有两种方法。一种方法是查询当前棋局中的每个可以放棋子的格子是否有 5 个格子的棋子相连并且是本方的棋子。当然，这种算法是对的，但不够简洁。另一种算法是直接判断最后下的那个棋子是否能和本方的棋子连成 5 个。本程序就使用这种判断方法来实现。

当然，游戏在决出胜负之后，不要忘记给游戏者一个提示框，让游戏者有机会保存棋局。

24.9 处理屏幕闪烁问题

▶ 本节视频教学录像：11 分钟

至此，程序代码基本完成。但我们运行程序的时候，发现屏幕总是会闪烁，让人炫目。这种现象是我们所不能容忍的，此时双缓冲技术应用而生。双缓冲技术在手机游戏中用的是最多的，原因是手机的内存相对较小，屏幕闪烁问题比较明显。

屏幕闪烁的原因：如果内存较小，运行速度较慢，当程序在绘制界面时，不是绘制的整个一张图，而是绘制的很多直线、曲线和各种填充等，这些绘制过程可能是成千上万个绘制动作，屏幕来不及显示，就会形成闪烁。Java 已经替我们做好了一个类，叫做 `BufferImage`。这个技术可避免形成屏幕闪烁。原理是不把成千上万条线逐一绘制到屏幕上去，而是绘制到缓冲区中，在需要显示界面的时候，直接显示缓冲区中的图片即可。

24.10 实现各个功能按钮

▶ 本节视频教学录像：1 小时

开始游戏：重新开始新的游戏。

游戏设置：设置倒计时。

游戏说明：用来说明游戏规则和操作。

认输：表示某一方放弃游戏，投子认输。

关于：用来显示程序的作者或编写单位的相关信息。

退出：结束程序。

24.11 完整代码

▶ 本节视频教学录像：28 分钟

本节给出本程序主类的完整代码。为了方便读者阅读，这里加上了目录和注释。

24.11.1 导入部分

```
01 package org.liky.game.frame;
02 import java.awt.Color;
03 import java.awt.Font;
04 import java.awt.Graphics;
```

```
05 import java.awt.Toolkit;
06 import java.awt.event.MouseEvent;
07 import java.awt.event.MouseListener;
08 import java.awt.image.BufferedImage;
09 import java.io.File;
10 import java.io.IOException;
11 import javax.imageio.ImageIO;
12 import javax.swing.JFrame;
13 import javax.swing.JOptionPane;
14 public class FiveChessFrame extends JFrame implements MouseListener, Runnable {
```

24.11.2 属性设置

```
01 // 取得屏幕的宽度
02 int width = Toolkit.getDefaultToolkit().getScreenSize().width;
03 // 取得屏幕的高度
04 int height = Toolkit.getDefaultToolkit().getScreenSize().height;
05 // 背景图片
06 BufferedImage bgImage = null;
07 // 保存棋子的坐标
08 int x = 0;
09 int y = 0;
10 // 保存之前下过的全部棋子的坐标
11 // 其中数据内容 0: 表示这个点并没有棋子, 1: 表示这个点是黑子, 2: 表示这个点是白子
12 int[][] allChess = new int[19][19];
13 // 标识当前应该黑棋还是白棋下一步
14 boolean isBlack = true;
15 // 标识当前游戏是否可以继续
16 boolean canPlay = true;
17 // 保存显示的提示信息
18 String message = "黑方先行";
19 // 保存最多拥有多少时间(秒)
20 int maxTime = 0;
21 // 做倒计时的线程类
22 Thread t = new Thread(this);
23 // 保存黑方与白方的剩余时间
24 int blackTime = 0;
25 int whiteTime = 0;
26 // 保存双方剩余时间的显示信息
27 String blackMessage = "无限制";
```

```
28      String whiteMessage = "无限制";
```

24.11.3 主类的构造函数

```
01      public FiveChessFrame() {
02          // 设置标题
03          this.setTitle("五子棋");
04          // 设置窗体大小
05          this.setSize(500, 500);
06          // 设置窗体出现位置
07          this.setLocation((width - 500) / 2, (height - 500) / 2);
08          // 将窗体设置为大小不可改变
09          this.setResizable(false);
10          // 将窗体的关闭方式设置为默认关闭后程序结束
11          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12          // 为窗体加入监听器
13          this.addMouseListener(this);
14          // 将窗体显示出来
15          this.setVisible(true);
16          t.start();
17          t.suspend();
18          // 刷新屏幕，防止开始游戏时出现无法显示的情况
19          this.repaint();
20          String imagePath = "";
21          try {
22              imagePath = System.getProperty("user.dir")+"bin/image/background.jpg";
23              bgImage = ImageIO.read(new File(imagePath.replaceAll("\\\\", "/")));
24          } catch (IOException e) {
25              // TODO Auto-generated catch block
26              e.printStackTrace();
27          }
28      }
```

24.11.4 Paint 方法

```
01      public void paint(Graphics g) {
02          // 双缓冲技术防止屏幕闪烁
03          BufferedImage bi = new BufferedImage(500, 500,
04              BufferedImage.TYPE_INT_RGB);
```

```
05     Graphics g2 = bi.createGraphics();
06     g2.setColor(Color.BLACK);
07     // 绘制背景
08     g2.drawImage(bgImage, 1, 20, this);
09     // 输出标题信息
10     g2.setFont(new Font("黑体", Font.BOLD, 20));
11     g2.drawString("游戏信息: " + message, 130, 60);
12     // 输出时间信息
13     g2.setFont(new Font("宋体", 0, 14));
14     g2.drawString("黑方时间: " + blackMessage, 30, 470);
15     g2.drawString("白方时间: " + whiteMessage, 260, 470);
16     // 绘制棋盘
17     for (int i = 0; i < 19; i++) {
18         g2.drawLine(10, 70 + 20 * i, 370, 70 + 20 * i);
19         g2.drawLine(10 + 20 * i, 70, 10 + 20 * i, 430);
20     }
21     // 标注点位
22     g2.fillOval(68, 128, 4, 4);
23     g2.fillOval(308, 128, 4, 4);
24     g2.fillOval(308, 368, 4, 4);
25     g2.fillOval(68, 368, 4, 4);
26     g2.fillOval(308, 248, 4, 4);
27     g2.fillOval(188, 128, 4, 4);
28     g2.fillOval(68, 248, 4, 4);
29     g2.fillOval(188, 368, 4, 4);
30     g2.fillOval(188, 248, 4, 4);
31     /*
32     * //绘制棋子 x = (x - 10) / 20 * 20 + 10; y = (y - 70) / 20 * 20 + 70;
33     * //黑子 g.fillOval(x - 7, y - 7, 14, 14); //白子 g.setColor(Color.WHITE);
34     * g.fillOval(x - 7, y - 7, 14, 14); g.setColor(Color.BLACK);
35     * g.drawOval(x - 7, y - 7, 14, 14);
36     */
37     // 绘制全部棋子
38     for (int i = 0; i < 19; i++) {
39         for (int j = 0; j < 19; j++) {
40             if (allChess[i][j] == 1) {
41                 // 黑子
42                 int tempX = i * 20 + 10;
43                 int tempY = j * 20 + 70;
44                 g2.fillOval(tempX - 7, tempY - 7, 14, 14);
45             }

```



```

46         if (allChess[i][j] == 2) {
47             // 白子
48             int tempX = i * 20 + 10;
49             int tempY = j * 20 + 70;
50             g2.setColor(Color.WHITE);
51             g2.fillOval(tempX - 7, tempY - 7, 14, 14);
52             g2.setColor(Color.BLACK);
53             g2.drawOval(tempX - 7, tempY - 7, 14, 14);
54         }
55     }
56 }
57 g.drawImage(bi, 0, 0, this);
58 }

```

24.11.5 监控鼠标

```

01 public void mouseClicked(MouseEvent e) {
02     // TODO Auto-generated method stub
03 }
04 public void mouseEntered(MouseEvent e) {
05     // TODO Auto-generated method stub
06 }
07 public void mouseExited(MouseEvent e) {
08     // TODO Auto-generated method stub
09 }
10 public void mousePressed(MouseEvent e) {
11     // TODO Auto-generated method stub
12     /*
13      * System.out.println("X:"+e.getX()); System.out.println("Y:"+e.getY());
14      */
15     if (canPlay == true) {
16         x = e.getX();
17         y = e.getY();
18         if (x >= 10 && x <= 370 && y >= 70 && y <= 430) {
19             x = (x - 10) / 20;
20             y = (y - 70) / 20;
21             if (allChess[x][y] == 0) {
22                 // 判断当前要下的是什么颜色的棋子
23                 if (isBlack == true) {
24                     allChess[x][y] = 1;

```

```
25         isBlack = false;
26         message = "轮到白方";
27     } else {
28         allChess[x][y] = 2;
29         isBlack = true;
30         message = "轮到黑方";
31     }
32     // 判断这个棋子是否和其他的棋子连成 5 连, 即判断游戏是否结束
33     boolean winFlag = this.checkWin();
34     if (winFlag == true) {
35         JOptionPane.showMessageDialog(this, "游戏结束,"
36             + (allChess[x][y] == 1 ? "黑方" : "白方") + "获胜! ");
37         canPlay = false;
38     }
39     } else {
40         JOptionPane.showMessageDialog(this, "当前位置已经有棋子, 请重新落子!");
41     }
42     this.repaint();
43 }
44 }
45 /* System.out.println(e.getX() + " -- " + e.getY()); */
46 // 单击 "开始游戏" 按钮
47 if (e.getX() >= 400 && e.getX() <= 470 && e.getY() >= 70
48     && e.getY() <= 100) {
49     int result = JOptionPane.showConfirmDialog(this, "是否重新开始游戏?");
50     if (result == 0) {
51         // 现在重新开始游戏
52         // 重新开始所要做的操作:(1) 把棋盘清空, allChess 这个数组中全部数据归 0
53         // (2) 将 "游戏信息:" 的显示改回到开始位置
54         // (3) 将下一步下棋的改为黑方
55         for (int i = 0; i < 19; i++) {
56             for (int j = 0; j < 19; j++) {
57                 allChess[i][j] = 0;
58             }
59         }
60         // 另一种方式 allChess = new int[19][19];
61         message = "黑方先行";
62         isBlack = true;
63         blackTime = maxTime;
64         whiteTime = maxTime;
65         if (maxTime > 0) {
```

```

66         blackMessage = maxTime / 3600 + ":"
67         + (maxTime / 60 - maxTime / 3600 * 60) + ":"
68         + (maxTime - maxTime / 60 * 60);
69         whiteMessage = maxTime / 3600 + ":"
70         + (maxTime / 60 - maxTime / 3600 * 60) + ":"
71         + (maxTime - maxTime / 60 * 60);
72         t.resume();
73     } else {
74         blackMessage = "无限制";
75         whiteMessage = "无限制";
76     }
77     this.canPlay = true;
78     this.repaint();
79 }
80 }
81 // 单击“游戏设置”按钮
82 if (e.getX() >= 400 && e.getX() <= 470 && e.getY() >= 120
83     && e.getY() <= 150) {
84     String input = JOptionPane
85         .showInputDialog("请输入游戏的最大时间(单位:分钟),如果输入 0,表示没有时间限制:");
86     try {
87         maxTime = Integer.parseInt(input) * 60;
88         if (maxTime < 0) {
89             JOptionPane.showMessageDialog(this, "请输入正确信息,不允许输入负数!");
90         }
91         if (maxTime == 0) {
92             int result = JOptionPane.showConfirmDialog(this,
93                 "设置完成,是否重新开始游戏?");
94             if (result == 0) {
95                 for (int i = 0; i < 19; i++) {
96                     for (int j = 0; j < 19; j++) {
97                         allChess[i][j] = 0;
98                     }
99                 }
100                 // 另一种方式 allChess = new int[19][19];
101                 message = "黑方先行";
102                 isBlack = true;
103                 blackTime = maxTime;
104                 whiteTime = maxTime;
105                 blackMessage = "无限制";
106                 whiteMessage = "无限制";

```

```
107         this.canPlay = true;
108         this.repaint();
109     }
110 }
111 if (maxTime > 0) {
112     int result = JOptionPane.showConfirmDialog(this,
113         "设置完成,是否重新开始游戏?");
114     if (result == 0) {
115         for (int i = 0; i < 19; i++) {
116             for (int j = 0; j < 19; j++) {
117                 allChess[i][j] = 0;
118             }
119         }
120         // 另一种方式 allChess = new int[19][19];
121         message = "黑方先行";
122         isBlack = true;
123         blackTime = maxTime;
124         whiteTime = maxTime;
125         blackMessage = maxTime / 3600 + ":"
126             + (maxTime / 60 - maxTime / 3600 * 60) + ":"
127             + (maxTime - maxTime / 60 * 60);
128         whiteMessage = maxTime / 3600 + ":"
129             + (maxTime / 60 - maxTime / 3600 * 60) + ":"
130             + (maxTime - maxTime / 60 * 60);
131         t.resume();
132         this.canPlay = true;
133         this.repaint();
134     }
135 }
136 } catch (NumberFormatException e1) {
137     // TODO Auto-generated catch block
138     JOptionPane.showMessageDialog(this, "请正确输入信息!");
139 }
140 }
141 // 单击“游戏说明”按钮
142 if (e.getX() >= 400 && e.getX() <= 470 && e.getY() >= 170
143     && e.getY() <= 200) {
144     JOptionPane.showMessageDialog(this,
145         "这是一个五子棋游戏程序,黑白双方轮流下棋,当某一方连到五子时,游戏结束。");
146 }
147 // 单击“认输”按钮
```

```

148         if (e.getX() >= 400 && e.getX() <= 470 && e.getY() >= 270
149             && e.getY() <= 300) {
150             int result = JOptionPane.showConfirmDialog(this, "是否确认认输?");
151             if (result == 0) {
152                 if (isBlack) {
153                     JOptionPane.showMessageDialog(this, "黑方已经认输,游戏结束!");
154                 } else {
155                     JOptionPane.showMessageDialog(this, "白方已经认输,游戏结束!");
156                 }
157                 canPlay = false;
158             }
159         }
160         // 单击“关于”按钮
161         if (e.getX() >= 400 && e.getX() <= 470 && e.getY() >= 320
162             && e.getY() <= 350) {
163             JOptionPane.showMessageDialog(this,
164                 "本游戏由 MLDN 制作, 有相关问题可以访问 www.mldn.cn");
165         }
166         // 单击“退出”按钮
167         if (e.getX() >= 400 && e.getX() <= 470 && e.getY() >= 370
168             && e.getY() <= 400) {
169             JOptionPane.showMessageDialog(this, "游戏结束");
170             System.exit(0);
171         }
172     }
173     public void mouseReleased(MouseEvent e) {
174         // TODO Auto-generated method stub
175     }

```

24.11.6 判断胜负

```

01     private boolean checkWin() {
02         boolean flag = false;
03         // 保存共有相同颜色多少棋子相连
04         int count = 1;
05         // 判断横向是否有 5 个棋子相连, 特点 纵坐标 是相同, 即 allChess[x][y]中 y 值是相同
06         int color = allChess[x][y];
07         /*
08         * if (color == allChess[x+1][y]) { count++; if (color ==
09         * allChess[x+2][y]) { count++; if (color == allChess[x+3][y]) {

```



```
10         * count++; } } }
11     */
12     // 通过循环来做棋子相连的判断
13     /*
14     * int i = 1; while (color == allChess[x + i][y + 0]) { count++; i++; }
15     * i = 1; while (color == allChess[x - i][y - 0]) { count++; i++; } if
16     * (count >= 5) { flag = true; } // 纵向的判断 int i2 = 1; int count2 = 1;
17     * while (color == allChess[x + 0][y + i2]) { count2++; i2++; } i2 = 1;
18     * while (color == allChess[x - 0][y - i2]) { count2++; i2++; } if
19     * (count2 >= 5) { flag = true; } // 斜方向的判断 (右上 + 左下) int i3 = 1; int
20     * count3 = 1; while (color == allChess[x + i3][y - i3]) { count3++;
21     * i3++; } i3 = 1; while (color == allChess[x - i3][y + i3]) { count3++;
22     * i3++; } if (count3 >= 5) { flag = true; } // 斜方向的判断 (右下 + 左上) int i4 =
23     * 1; int count4 = 1; while (color == allChess[x + i4][y + i4]) {
24     * count4++; i4++; } i4 = 1; while (color == allChess[x - i4][y - i4]) {
25     * count4++; i4++; } if (count4 >= 5) { flag = true; }
26     */
27     // 判断横向
28     count = this.checkCount(1, 0, color);
29     if (count >= 5) {
30         flag = true;
31     } else {
32         // 判断纵向
33         count = this.checkCount(0, 1, color);
34         if (count >= 5) {
35             flag = true;
36         } else {
37             // 判断右上、左下
38             count = this.checkCount(1, -1, color);
39             if (count >= 5) {
40                 flag = true;
41             } else {
42                 // 判断右下、左上
43                 count = this.checkCount(1, 1, color);
44                 if (count >= 5) {
45                     flag = true;
46                 }
47             }
48         }
49     }
50     return flag;
```

51 }

24.11.7 判断有几个棋子已经连接起来

```

01 // 判断棋子连接的数量
02 private int checkCount(int xChange, int yChange, int color) {
03     int count = 1;
04     int tempX = xChange;
05     int tempY = yChange;
06     while (x + xChange >= 0 && x + xChange <= 18 && y + yChange >= 0
07         && y + yChange <= 18
08         && color == allChess[x + xChange][y + yChange]) {
09         count++;
10         if (xChange != 0)
11             xChange++;
12         if (yChange != 0) {
13             if (yChange > 0)
14                 yChange++;
15             else {
16                 yChange--;
17             }
18         }
19     }
20     xChange = tempX;
21     yChange = tempY;
22     while (x - xChange >= 0 && x - xChange <= 18 && y - yChange >= 0
23         && y - yChange <= 18
24         && color == allChess[x - xChange][y - yChange]) {
25         count++;
26         if (xChange != 0)
27             xChange++;
28         if (yChange != 0) {
29             if (yChange > 0)
30                 yChange++;
31             else {
32                 yChange--;
33             }
34         }
35     }
36     return count;

```

```

37     }
38     public void run() {
39         // TODO Auto-generated method stub
40         // 判断是否有时间限制
41         if (maxTime > 0) {
42             while (true) {
43                 if (isBlack) {
44                     blackTime--;
45                     if (blackTime == 0) {
46                         JOptionPane.showMessageDialog(this, "黑方超时,游戏结束!");
47                     }
48                 } else {
49                     whiteTime--;
50                     if (whiteTime == 0) {
51                         JOptionPane.showMessageDialog(this, "白方超时,游戏结束!");
52                     }
53                 }
54                 blackMessage = blackTime / 3600 + ":"
55                     + (blackTime / 60 - blackTime / 3600 * 60) + ":"
56                     + (blackTime - blackTime / 60 * 60);
57                 whiteMessage = whiteTime / 3600 + ":"
58                     + (whiteTime / 60 - whiteTime / 3600 * 60) + ":"
59                     + (whiteTime - whiteTime / 60 * 60);
60                 this.repaint();
61                 try {
62                     Thread.sleep(1000);
63                 } catch (InterruptedException e) {
64                     // TODO Auto-generated catch block
65                     e.printStackTrace();
66                 }
67                 System.out.println(blackTime + " -- " + whiteTime);
68             }
69         }
70     }
71 }

```

第 25 章

Java项目开发实战——人事管理



本章视频教学录像：1 小时 8 分钟

通过前面章节的学习，相信读者已经对在Java中进行开发应用程序的过程比较熟悉了，本章通过一个人事管理系统的设计，深入学习Java的项目开发。本章内容包括系统需求分析方法、系统的开发步骤、数据库及接口设计等知识，通过本章知识的学习，读者将对开发一个Java项目的具体流程有一定的了解。

本章要点（已掌握的在方框中打勾）

- ☐ 掌握人事系统的需求分析方法
- ☐ 掌握人事系统的关键技术
- ☐ 掌握人事系统的数据库设计
- ☐ 掌握人事系统的接口设计



25.1 系统概述

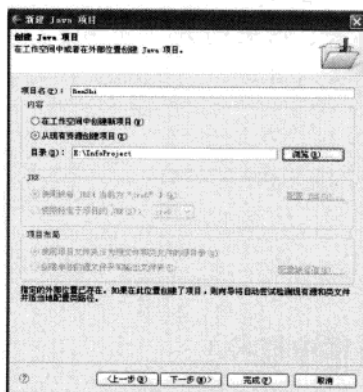
本系统是一个教学案例。读者可以先试运行本书提供的源码，以对需要开发的程序的功能有一个大致的了解。

25.1.1 运行系统

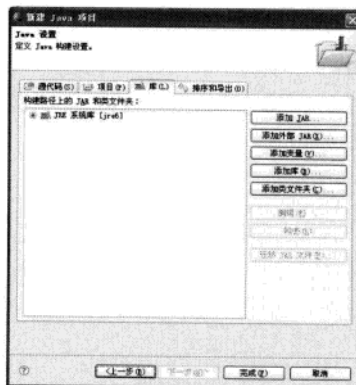
下面简述运行本系统的具体步骤。

第 1 步：创建 Java 项目

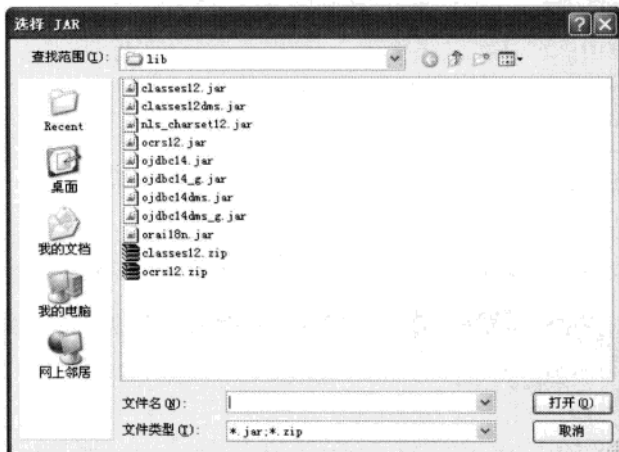
- ① 把光盘中的“final\ch25\InfoProject”目录复制到硬盘中，本例使用“E:\”。
- ② 运行 Eclipse，新建一个 Java 项目，本例命名为“RenShi”（注意：尽量不要命名为 InfoProject，与本书开发的系统重名，否则有可能造成以后不能新建以这个名字命名的项目）。勾选【从现有资源创建项目】单选按钮，在【目录】文本框中输入“E:\infoProject”，然后单击【下一步】按钮。



- ③ 在出现的【Java 设置】对话框中选择【库】选项卡。



- ④ 单击【添加外部 JAR】按钮，出现【选择 JAR】对话框。



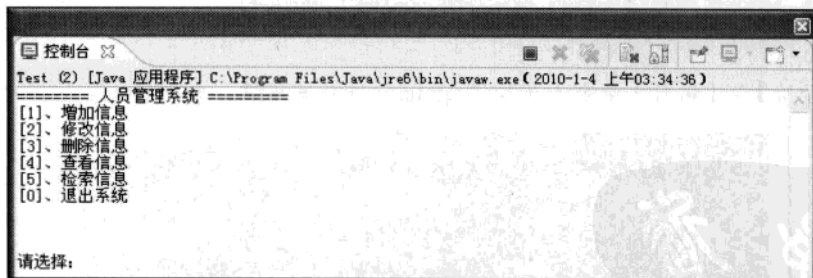
- ⑤ 选择 Oracle 数据库所提供的 jar 文件，本例中的路径为“E:\oracle\product\10.1.0\Db_1\jdbc\lib\classes12.jar”。



注意：如果使用的 jre 版本不对，也有可能造成本程序无法执行。请更换合适的 jre 版本设置环境变量，并在 Eclipse 中进行设置。

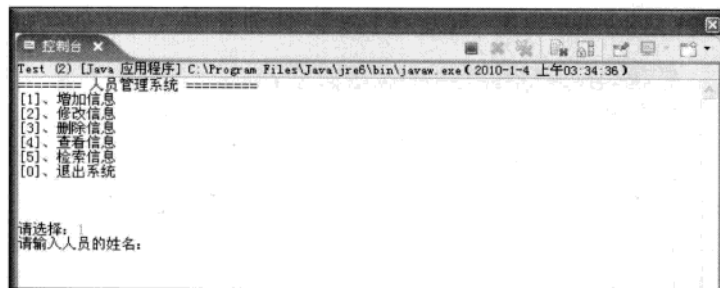
第 2 步：运行项目

- ① 在【包资源管理器】中选择【RenShi】>【src】>【org.lxx.info.test】>【Test.java】，右击该项，然后在弹出的快捷菜单中选择【运行方式】>【1 Java 应用程序】菜单项，即可在 Eclipse 的【控制台】窗口中出现人事管理的界面。

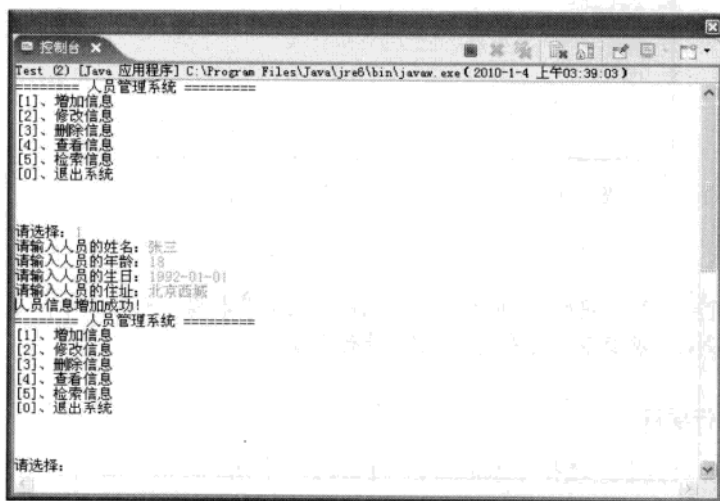


注意：如果数据库中的设置不是按照本书第 23 章进行的，可能还需要修改程序代码才能运行成功。本书的设置是：默认数据库名称为 orcl，用户 system 的密码是 java。修改程序代码的方法为：在【包资源管理器】中选择【RenShi】>【src】>【org.lxx.info.dbc】>【DatabaseConnection.java】以打开该文件，然后进行相应的修改即可。

- ② 输入“1”，然后按回车键，以增加信息。

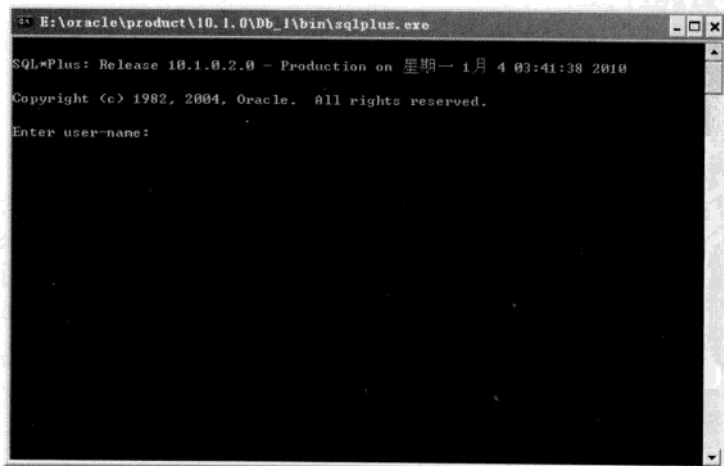


③ 依次输入姓名、年龄、生日和住址等信息，然后按回车键，即可在数据库中插入一条记录。

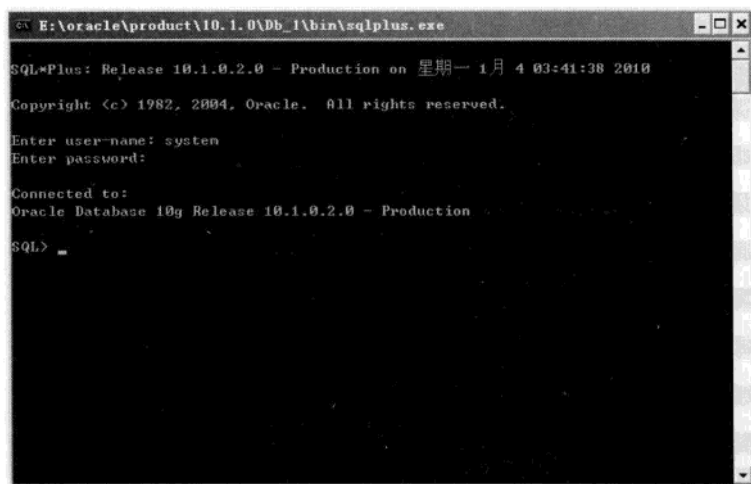


注意：如果输入的日期格式不正确，将会出现异常。

④ 选择【开始】>【运行】命令，输入“sqlplus”并按回车键，出现 Oracle 的客户端软件。



- ⑤ 输入用户名“system”和密码“java”登录数据库。

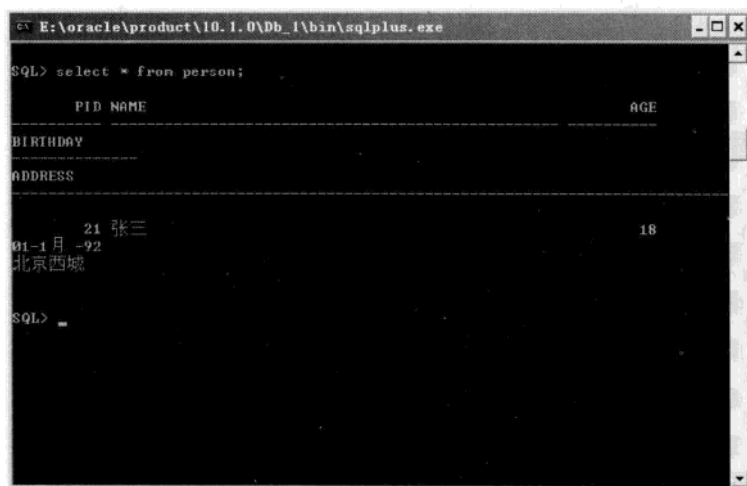


```

E:\oracle\product\10.1.0\Db_1\bin\sqlplus.exe
SQL*Plus: Release 10.1.0.2.0 - Production on 星期一 1月 4 03:41:38 2010
Copyright (c) 1982, 2004, Oracle. All rights reserved.
Enter user-name: system
Enter password:
Connected to:
Oracle Database 10g Release 10.1.0.2.0 - Production
SQL>

```

- ⑥ 输入“select * from person;”并按回车键。



```

SQL> select * from person;

      PID NAME                                AGE
-----
BIRTHDAY
ADDRESS
-----
      21 张三
01-1月 -92      18
北京西城

SQL>

```

可以看到“张三”的信息已经被添加到了数据库中。

25.1.2 系统的开发步骤

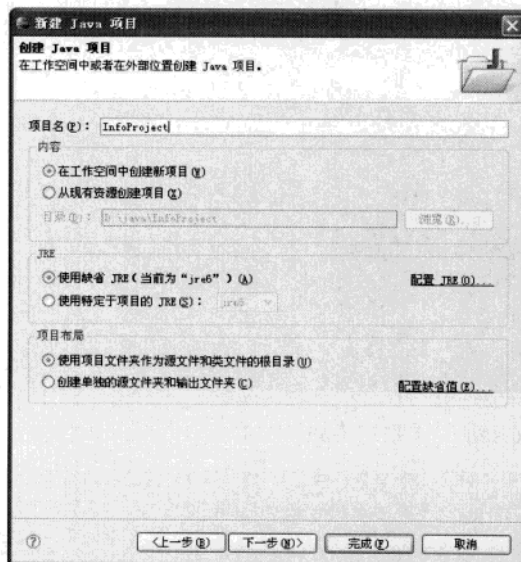
开发本程序的具体步骤如下。

- ① 新建项目，并新建一个测试类用于运行主程序。
- ② 新建一个类或者接口等，编写代码完成任务。
- ③ 测试代码。
- ④ 运行程序。

下面给出①和②的详细步骤。

① 建立新的 Java 项目 InfoProject。

这里的工作空间指定为：D:\java\ch25\final。



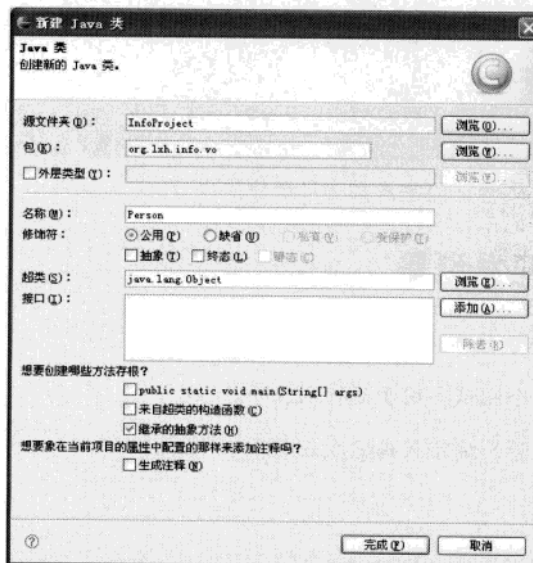
② 在该项目中创建一个类 Person。

其参数如下。

包：org.lxh.info.test

名称：Person

其他选项为默认设置即可。



- ③ 根据以下讲解分别创建各种类型，并书写代码。

25.2 系统需求分析

本系统是一个教学案例，所以它的功能非常简单，仅仅是对某单位员工进行简单的人事管理。可以对人员进行添加、修改，根据员工编号进行删除，浏览全部员工信息和查询员工信息等操作。

员工信息包括：员工编号、姓名、年龄、生日和通讯地址等。

25.3 综合描述

本系统的开发平台和工具如下。

- (1) JDK 1.6.0_17 多国语言版。
- (2) Eclipse 是 3.2.0 版，中文语言包。
- (3) Windows XP Professional Service Pack 3。
- (4) Oracle 10g。

25.3.1 关键技术

- (1) 设计模式。使用面向对象的方法进行合理的类的结构划分。
- (2) 使用 `BufferedReader` 或 `Scanner` 类完成信息的输入。
- (3) 使用 `SimpleDateFormat` 类进行日期格式的转换。
- (4) 使用 JDBC 技术进行数据库的操作。
- (5) 使用 Oracle 进行数据的保存，使用 `Sequence` 进行自动增长列的操作。
- (6) 使用类集框架进行数据的检索操作。

25.3.2 名词解释

设计模式 (Design pattern) 是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码，让代码更容易被他人理解，保证代码的可靠性。

毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式可使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。

25.3.3 运行环境

本系统可运行在安装了 JRE 1.6 和 Oracle 10g 的任何一个操作系统上。

25.4 概要设计

概要设计就是把大的方向确定下来，这样在后面实现具体代码时，就能有的放矢、合理安排，不会出现大的偏差。

25.4.1 数据库设计

开发系统首先要进行数据库的设计，不过本系统很简单，只需要设计一张表，仅仅用几条SQL语句即可。

```
01 DROP TABLE person ;
02 DROP SEQUENCE myseq ;
03 CREATE SEQUENCE myseq ;
04 CREATE TABLE person(
05     pid      NUMBER      PRIMARY KEY ,
06     name     VARCHAR2(50) NOT NULL ,
07     age      NUMBER      NOT NULL ,
08     birthday DATE        NOT NULL ,
09     address  VARCHAR2(200)
10 );
```

第一句话，是说如果当前数据库中存在表 person 的话，先删除这张表，否则后面的操作将会出错。

第二句话，是让数据库自动产生一个不重复的员工编号。

后面的代码，是进行数据库表格的详细设计。共 5 个字段，pid 存储员工编号，name 存储员工姓名，age 存储员工年龄，birthday 存储员工的生日，address 存储员工地址。

25.4.2 接口设计

可以将接口理解为定义方法的类，但这个类并不实现方法，由子类实现。和抽象类有点像，但抽象类可以实现方法。

```
01 public interface MyInterface
02 {
03     void Method1();
04     string Method2(int param);
05 }
```

以上是一个接口的定义，里面的 Method1 就是一个定义，但没有具体的实现。等有个类继承了这个接口，就必须实现里面的方法。

本程序需要设计一个接口，来表明本程序的具体功能。代码如下。

```
01 package org.lxh.info.dao;
02 import java.util.List;
03 import org.lxh.info.vo.Person;
04 public interface IPersonDAO {
05     /**
06      * 数据库的增加操作
07      * @param person
08      * @return
09      * @throws Exception
10      */
11     public boolean doCreate(Person person) throws Exception;
12     /**
13      * 数据库的修改操作
14      * @param person
15      * @return
16      * @throws Exception
17      */
18     public boolean doUpdate(Person person) throws Exception;
19     /**
20      * 数据库的删除操作
21      * @param pid
22      * @return
23      * @throws Exception
24      */
25     public boolean doDelete(int pid) throws Exception;
26     /**
27      * 根据 ID 查询数据库
28      * @param pid
29      * @return
30      * @throws Exception
31      */
32     public Person findById(int pid) throws Exception;
33     /**
34      * 查询全部的记录
35      * @param keyWord
36      * @return
37      * @throws Exception
38      */
```

```
39     public List<Person> findAll(String keyWord) throws Exception;  
40 }
```

以上的代码非常清晰，直接说明了本程序的功能：添加、修改、删除和查找记录。

25.4.3 代理

代理是一种设计模式。

举个例子：你想投资，但不懂股票，于是就去买了个基金，你只需要关注这个基金的涨跌，至于这个基金买了什么股票，什么时候买的，这些你都不需要知道。

但这个基金必须有一定的接口，就是必须有一定的服务标准（譬如必须收取一些委托信函，返回一个收据给你）。这就是代理模式和接口的关系。

只有学好了设计模式，才能灵活使用 Java 语言编写出简洁、高效、健壮、可重用性和可修改性高的代码。下面仅仅介绍一下设计模式的分类。

1. 创建型设计模式

- (1) 工厂方法（Factory Method）模式
- (2) 抽象工厂（Abstract Factory）模式
- (3) 原型（Prototype）模式
- (4) 单例（Singleton）模式
- (5) 建造（Builder）模式等

2. 结构型设计模式

- (1) 合成（Composite）模式
- (2) 装饰（Decorator）模式
- (3) 代理（Proxy）模式
- (4) 享元（Flyweight）模式
- (5) 门面（Facade）模式
- (6) 桥梁（Bridge）模式等

3. 行为型模式

- (1) 模板方法（Template Method）模式
- (2) 观察者（Observer）模式
- (3) 迭代子（Iterator）模式
- (4) 责任链（Chain of Responsibility）模式
- (5) 备忘录（Memento）模式
- (6) 命令（Command）模式
- (7) 状态（State）模式
- (8) 访问者（Visitor）模式

25.5 代码实现

有了上面的分析，本节书写具体的代码来完成人员管理系统。

25.5.1 Person.java

首先要创建一个类，把数据库表格中的数据统统都装到这个类生成的对象里面，或者说数据库中的表的各个字段和这个类的属性的各个成员一一对应。具体代码如下。

```
01 package org.lxh.info.vo;
02 import java.util.Date;
03 public class Person {           // 本类可以很好地映射一个表的数据
04     private int pid;            // 对应员工编号字段
05     private String name;        // 对应员工姓名字段
06     private int age;            // 对应员工年龄字段
07     private Date birthday;      // 对应员工生日字段
08     private String address;     // 对应员工地址字段
09
10     public Person() {
11         super();
12     }
13     public Person(int pid, String name, int age, Date birthday, String address) {
14         // 构造方法，要传入 5 个参数对应 5 个字段
15         super();
16         this.pid = pid;
17         this.name = name;
18         this.age = age;
19         this.birthday = birthday;
20         this.address = address;
21     }
22     // 以下代码仅仅是为了完成属性的封装
23     public int getPid() {
24         return pid;
25     }
26     public void setPid(int pid) {
27         this.pid = pid;
28     }
29     public String getName() {
30         return name;
31     }
```

```

32     public void setName(String name) {
33         this.name = name;
34     }
35     public int getAge() {
36         return age;
37     }
38     public void setAge(int age) {
39         this.age = age;
40     }
41     public Date getBirthday() {
42         return birthday;
43     }
44     public void setBirthday(Date birthday) {
45         this.birthday = birthday;
46     }
47     public String getAddress() {
48         return address;
49     }
50     public void setAddress(String address) {
51         this.address = address;
52     }
53
54 }

```

25.5.2 IPersonDAO.java

在一个项目的开发中首先必须完成的是接口的设计。接口需要完成所有需求分析中所提到的功能。

一张数据库的表类似于一个类,我们现在已经建立了一个类与数据库中的表的各个字段进行对应。那么,现在为了更好地表示出与数据库中的表的关系,仅用一个类表示还不够。现在操作的是数据,所以接口的主要目的就是完成数据的操作。那么接口的所有对象,也都是数据的操作对象。可以使用 DataAccessObject 表示,简称 DAO。即一个 DAO 表示一个数据的操作对象。

因为操作的是一张 Person 表,因此所有接口的名称为 IPersonDAO。

代码如下。

```

01     package org.lxx.info.dao;
02     import java.util.List;
03     import org.lxx.info.vo.Person;
04     public interface IPersonDAO {
05         /**
06         * 数据库的增加操作

```



```

07      * @param person
08      * @return
09      * @throws Exception
10      */
11      public boolean doCreate(Person person) throws Exception;
12      /**
13       * 数据库的修改操作
14       * @param person
15       * @return
16       * @throws Exception
17       */
18      public boolean doUpdate(Person person) throws Exception;
19      /**
20       * 数据库的删除操作
21       * @param pid
22       * @return
23       * @throws Exception
24       */
25      public boolean doDelete(int pid) throws Exception;
26      /**
27       * 根据 ID 查询数据库
28       * @param pid
29       * @return
30       * @throws Exception
31       */
32      public Person findById(int pid) throws Exception;
33      /**
34       * 查询全部的记录
35       * @param keyWord
36       * @return
37       * @throws Exception
38       */
39      public List<Person> findAll(String keyWord) throws Exception;
40  }

```

25.5.3 DatabaseConnection.java

凡是要进行数据库操作，必须要进行数据库连接和关闭数据库的操作。这些操作有以下 3 个特点。

- (1) 这些操作是每访问一次数据库都必须做的。

(2) 这些操作对具体目标的达成都没有直接的逻辑上的联系，仅仅是编程所需要的，和具体业务没有关系。

(3) 这些操作是普遍重复使用的。

鉴于此，我们可以把这些枯燥的操作做成一个单独的类，当需要的时候就调用它来完成，以分清业务的主次。

代码如下。

```
01 package org.lxh.info.dbc;
02 import java.sql.Connection;
03 import java.sql.DriverManager;
04 import java.sql.SQLException;
05 public class DatabaseConnection {
06     public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
07     public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:orcl";
08     public static final String DBUSER = "system";
09     public static final String DBPASSWORD = "java";
10     private Connection conn = null;
11     public DatabaseConnection() {
12         try {
13             Class.forName(DBDRIVER);
14             this.conn = DriverManager.getConnection(DBURL, DBUSER, DBPASSWORD);
15         } catch (ClassNotFoundException e) {
16             e.printStackTrace();
17         } catch (SQLException e) {
18             e.printStackTrace();
19         }
20     }
21     public Connection getConnection() {
22         return this.conn;
23     }
24     public void close() {
25         if (this.conn != null) {
26             try {
27                 this.conn.close();
28             } catch (SQLException e) {
29                 e.printStackTrace();
30             }
31         }
32     }
33 }
```

25.5.4 IPersonDAOProxy.java

使用代理是设计模式的思想。代理其实并不做真正的业务，只是安排业务而已。但对工作的条理化，工作的顺利完成，则意义重大。

利用数据库代理来进行分工，制定详细的“游戏规则”。

(1) 关于数据库的打开和关闭等操作这些“脏活累活”，由 DatabaseConnection 类来完成。

(2) 关于具体业务由 IPersonDAOImpl 来完成。

具体代码如下。

```
01 package org.lxh.info.dao.proxy;
02 import java.util.List;
03 import org.lxh.info.dao.IPersonDAO;
04 import org.lxh.info.dao.impl.IPersonDAOImpl;
05 import org.lxh.info.dbc.DatabaseConnection;
06 import org.lxh.info.vo.Person;
07 public class IPersonDAOProxy implements IPersonDAO {
08     private DatabaseConnection dbc = null;
09     private IPersonDAO dao = null;
10     public IPersonDAOProxy() {
11         this.dbc = new DatabaseConnection();
12         this.dao = new IPersonDAOImpl(this.dbc.getConnection());
13     }
14     public boolean doCreate(Person person) throws Exception {
15         boolean flag = false;
16         try {
17             flag = this.dao.doCreate(person);
18         } catch (Exception e) {
19             throw e;
20         } finally {
21             this.dbc.close();
22         }
23         return flag;
24     }
25     public boolean doDelete(int pid) throws Exception {
26         boolean flag = false;
27         try {
28             flag = this.dao.doDelete(pid);
29         } catch (Exception e) {
30             throw e;
31         } finally {
32             this.dbc.close();
33         }
34         return flag;
35     }
36     public List doFind() throws Exception {
37         List list = null;
38         try {
39             list = this.dao.doFind();
40         } catch (Exception e) {
41             throw e;
42         } finally {
43             this.dbc.close();
44         }
45         return list;
46     }
47     public boolean doUpdate(Person person) throws Exception {
48         boolean flag = false;
49         try {
50             flag = this.dao.doUpdate(person);
51         } catch (Exception e) {
52             throw e;
53         } finally {
54             this.dbc.close();
55         }
56         return flag;
57     }
58 }
```

```

32         this.dbc.close();
33     }
34     return flag;
35 }
36 public boolean doUpdate(Person person) throws Exception {
37     boolean flag = false;
38     try {
39         flag = this.dao.doUpdate(person);
40     } catch (Exception e) {
41         throw e;
42     } finally {
43         this.dbc.close();
44     }
45     return flag;
46 }
47 public List<Person> findAll(String keyWord) throws Exception {
48     List<Person> all = null;
49     try {
50         all = this.dao.findAll(keyWord);
51     } catch (Exception e) {
52         throw e;
53     } finally {
54         this.dbc.close();
55     }
56     return all;
57 }
58 public Person findById(int pid) throws Exception {
59     Person per = null;
60     try {
61         per = this.dao.findById(pid);
62     } catch (Exception e) {
63         throw e;
64     } finally {
65         this.dbc.close();
66     }
67     return per;
68 }
69 }

```

25.5.5 IPersonDAOImpl.java

在 Java 世界里有严格的等级制度，IPersonDAOImpl 接到了代理（其实相当于中介公司）分派的任务，就该老老实实在地实现它自己具体业务的代码，这也是本程序的核心业务代码。

具体代码如下。

```
01 package org.lxh.info.dao.impl;
02 import java.sql.Connection;
03 import java.sql.PreparedStatement;
04 import java.sql.ResultSet;
05 import java.util.ArrayList;
06 import java.util.List;
07 import org.lxh.info.dao.IPersonDAO;
08 import org.lxh.info.vo.Person;
09 public class IPersonDAOImpl implements IPersonDAO {
10     private Connection conn = null;
11     public IPersonDAOImpl(Connection conn) {
12         this.conn = conn;
13     }
14     public boolean doCreate(Person person) throws Exception {
15         boolean flag = false;
16         PreparedStatement pstmt = null;
17         String sql = "INSERT INTO person(pid,name,age,birthday,address)"
18             + " VALUES (myseq.nextval,?,?,?,?)";
19         try {
20             pstmt = this.conn.prepareStatement(sql);
21             pstmt.setString(1, person.getName());
22             pstmt.setInt(2, person.getAge());
23             pstmt.setDate(3, new java.sql.Date(person.getBirthday().getTime()));
24             pstmt.setString(4, person.getAddress());
25             int len = pstmt.executeUpdate();
26             if (len > 0) {
27                 flag = true;
28             }
29         } catch (Exception e) {
30             throw e;
31         } finally {
32             try {
33                 pstmt.close();
34             } catch (Exception e) {
35                 throw e;
36             }
37         }
38     }
39 }
```



```
36     }
37 }
38     return flag;
39 }
40 public boolean doDelete(int pid) throws Exception {
41     boolean flag = false;
42     PreparedStatement pstmt = null;
43     String sql = "DELETE FROM person WHERE pid=?";
44     try {
45         pstmt = this.conn.prepareStatement(sql);
46         pstmt.setInt(1, pid);
47         int len = pstmt.executeUpdate();
48         if (len > 0) {
49             flag = true;
50         }
51     } catch (Exception e) {
52         throw e;
53     } finally {
54         try {
55             pstmt.close();
56         } catch (Exception e) {
57             throw e;
58         }
59     }
60     return flag;
61 }
62 public boolean doUpdate(Person person) throws Exception {
63     boolean flag = false;
64     PreparedStatement pstmt = null;
65     String sql = "UPDATE person SET name=?,age=?,birthday=?,address=? WHERE pid=?";
66     try {
67         pstmt = this.conn.prepareStatement(sql);
68         pstmt.setString(1, person.getName());
69         pstmt.setInt(2, person.getAge());
70         pstmt.setDate(3, new java.sql.Date(person.getBirthday().getTime()));
71         pstmt.setString(4, person.getAddress());
72         pstmt.setInt(5, person.getPid());
73         int len = pstmt.executeUpdate();
74         if (len > 0) {
75             flag = true;
76         }
77     }
```

```
77         } catch (Exception e) {
78             throw e;
79         } finally {
80             try {
81                 pstmt.close();
82             } catch (Exception e) {
83                 throw e;
84             }
85         }
86         return flag;
87     }
88     public List<Person> findAll(String keyWord) throws Exception {
89         List<Person> all = new ArrayList<Person>();
90         PreparedStatement pstmt = null;
91         String sql = "SELECT pid,name,age,birthday,address FROM person "
92             + "WHERE name LIKE ? OR age LIKE ? OR birthday LIKE ? OR address LIKE ?";
93         try {
94             pstmt = this.conn.prepareStatement(sql);
95             pstmt.setString(1, "%" + keyWord + "%"); // 模糊查询
96             pstmt.setString(2, "%" + keyWord + "%"); // 模糊查询
97             pstmt.setString(3, "%" + keyWord + "%"); // 模糊查询
98             pstmt.setString(4, "%" + keyWord + "%"); // 模糊查询
99             ResultSet rs = pstmt.executeQuery(); // 执行查询
100             Person per = null;
101             while (rs.next()) { // 如果有查询的结果,则可以向下执行
102                 per = new Person();
103                 per.setPid(rs.getInt(1));
104                 per.setName(rs.getString(2));
105                 per.setAge(rs.getInt(3));
106                 per.setBirthday(rs.getDate(4));
107                 per.setAddress(rs.getString(5));
108                 all.add(per); // 向集合中插入内容
109             }
110         } catch (Exception e) {
111             throw e;
112         } finally {
113             try {
114                 pstmt.close();
115             } catch (Exception e) {
116                 throw e;
117             }
118         }
119     }
120 }
```

```

118     }
119     return all;
120 }
121
122
123 public Person findByd(int pid) throws Exception {
124     Person per = null;
125     PreparedStatement pstmt = null;
126     String sql = "SELECT pid,name,age,birthday,address FROM person WHERE pid=?";
127     try {
128         pstmt = this.conn.prepareStatement(sql);
129         pstmt.setInt(1, pid);
130         ResultSet rs = pstmt.executeQuery(); // 执行查询
131         if (rs.next()) { // 如果有查询的结果，则可以向下执行
132             per = new Person();
133             per.setPid(rs.getInt(1));
134             per.setName(rs.getString(2));
135             per.setAge(rs.getInt(3));
136             per.setBirthday(rs.getDate(4));
137             per.setAddress(rs.getString(5));
138         }
139     } catch (Exception e) {
140         throw e;
141     } finally {
142         try {
143             pstmt.close();
144         } catch (Exception e) {
145             throw e;
146         }
147     }
148     return per;
149 }
150 }

```

25.5.6 DAOFactory.java

工厂也是设计模式中的一个重要概念。你只要使用了代理，程序中就会出现接口，这时需要进行解耦合操作，这就是工厂。凡是出现接口，想实例化了，一般都要使用工厂。

```

01 package org.lxx.info.factory;
02 import org.lxx.info.dao.IPersonDAO;

```

```
03 import org.lxh.info.dao.proxy.IPersonDAOProxy;
04 public class DAOFactory {
05     public static IPersonDAO getIPersonDAOInstance() {
06         return new IPersonDAOProxy();
07     }
08 }
```

25.5.7 Menu.java

至此，程序的数据层已经完成。

下面需要进行前台界面的设计，首先是菜单设计。代码如下。

```
01 package org.lxh.info.menu;
02 import org.lxh.info.operate.PersonOperate;
03 import org.lxh.info.util.InputData;
04 public class Menu {
05     public Menu() {
06         while (true) {
07             this.show();
08         }
09     }
10     public void show() { // 显示菜单
11         System.out.println("===== 人员管理系统 =====");
12         System.out.println("[1]、增加信息");
13         System.out.println("[2]、修改信息");
14         System.out.println("[3]、删除信息");
15         System.out.println("[4]、查看信息");
16         System.out.println("[5]、检索信息");
17         System.out.println("[0]、退出系统\n\n\n");
18         int ch = new InputData().getInt("请选择：", "选项必须是数字");
19         switch (ch) {
20             case 0: {
21                 System.out.println("bye bye.");
22                 System.exit(1);
23             }
24             case 1: {
25                 PersonOperate.insert();
26                 break;
27             }
28             case 2: {
29                 PersonOperate.update();
```

```

30         break;
31     }
32     case 3: {
33         PersonOperate.delete();
34         break;
35     }
36     case 4: {
37         PersonOperate.findall();
38         break;
39     }
40     case 5: {
41         PersonOperate.search();
42         break;
43     }
44     default: {
45         System.out.println("请选择正确的选项。");
46     }
47 }
48 }
49 }

```

25.5.8 InputData.java

菜单程序写完了，下面需要输入数据，接受用户数据。
代码如下。

```

01 package org.lxh.info.util;
02 import java.text.ParseException;
03 import java.text.SimpleDateFormat;
04 import java.util.Date;
05 import java.util.Scanner;
06 public class InputData {
07     private Scanner scan = null;    // 输入数据
08     public InputData() {
09         this.scan = new Scanner(System.in);    // 输入数据实例化
10     }
11     public String getString(String info) {
12         String str = null;
13         System.out.print(info);
14         if (this.scan.hasNext()) {
15             str = this.scan.next();    // 接收内容

```



```

16     }
17     return str;
18 }
19 public int getInt(String info, String errMsg) {
20     int temp = 0;
21     System.out.print(info);
22     if (this.scan.hasNextInt()) {
23         temp = this.scan.nextInt();    // 接收内容
24     } else {
25         System.out.print(errMsg);
26     }
27     return temp;
28 }
29 public Date getDate(String info, String errMsg) {
30     java.util.Date date = null;
31     System.out.print(info);
32     if (this.scan.hasNext("\\d{4}-\\d{2}-\\d{2}")) {
33         String str = this.scan.next("\\d{4}-\\d{2}-\\d{2}");
34         try {
35             date = new SimpleDateFormat("yyyy-MM-dd").parse(str);
36         } catch (ParseException e) {
37             e.printStackTrace();
38         }
39     } else {
40         System.out.print(errMsg);
41     }
42     return date;
43 }
44 }

```

25.5.9 PersonOperate.java

显示了菜单，接收了数据，接下来要对这些数据进行处理，然后交由工厂进行处理。下面的程序显示了如何对接收到的数据进行处理，然后转到工厂进行加工，进行相应的操作。代码如下。

```

01 package org.lxh.info.operate;
02 import java.util.Date;
03 import java.util.Iterator;
04 import java.util.List;
05 import org.lxh.info.factory.DAOFactory;

```

```
06 import org.lxx.info.util.InputData;
07 import org.lxx.info.vo.Person;
08 public class PersonOperate {
09     public static void insert() {
10         Person per = new Person();
11         InputData input = new InputData();
12         String name = input.getString("请输入人员的姓名: ");
13         int age = input.getInt("请输入人员的年龄: ", "年龄必须是数字, ");
14         Date date = input.getDate("请输入人员的生日: ", "输入的日期格式不正确, ");
15         String address = input.getString("请输入人员的住址: ");
16         per.setName(name);
17         per.setAddress(address);
18         per.setAge(age);
19         per.setBirthday(date);
20         try {
21             DAOFactory.getIPersonDAOInstance().doCreate(per);
22             System.out.println("人员信息增加成功! ");
23         } catch (Exception e) {
24             System.out.println("人员信息增加失败! ");
25         }
26     }
27     public static void update() {
28         // 在修改数据之前最好先将数据查询出来
29         Person per = null;
30         InputData input = new InputData();
31         int pid = input.getInt("请输入要修改人员的编号: ", "编号必须是数字, ");
32         try {
33             per = DAOFactory.getIPersonDAOInstance().findById(pid);
34         } catch (Exception e1) {
35         }
36         if (per != null) {
37             String name = input.getString("请输入新的人员的姓名 (原姓名是: " + per.getName()
38                 + "): ");
39             int age = input.getInt("请输入新的人员的年龄 (原年龄是: " + per.getAge() + "): ",
40                 "年龄必须是数字, ");
41             Date date = input.getDate("请输入新的人员的生日 (原生日是: " + per.getBirthday()
42                 + "): ", "输入的日期格式不正确, ");
43             String address = input.getString("请输入新的人员的住址 (原住址是: "
44                 + per.getAddress() + "): ");
45             per.setName(name);
46             per.setAddress(address);
```

```
47         per.setAge(age);
48         per.setBirthday(date);
49         try {
50             DAOFactory.getIPersonDAOInstance().doUpdate(per);
51             System.out.println("人员信息修改成功！");
52         } catch (Exception e) {
53             System.out.println("人员信息修改失败！");
54         }
55     } else {
56         System.out.println("没有此人员信息。");
57     }
58 }
59 public static void delete() {
60     InputData input = new InputData();
61     int pid = input.getInt("请输入要修改人员的编号：", "编号必须是数字，");
62     try {
63         DAOFactory.getIPersonDAOInstance().doDelete(pid);
64         System.out.println("人员信息删除成功！");
65     } catch (Exception e) {
66         System.out.println("人员信息删除失败！");
67     }
68 }
69 public static void findall() {
70     List<Person> all = null;
71     try {
72         all = DAOFactory.getIPersonDAOInstance().findAll("");
73     } catch (Exception e) {
74         e.printStackTrace();
75     }
76     Iterator<Person> iter = all.iterator();
77     while (iter.hasNext()) {
78         Person per = iter.next();
79         System.out.println(per.getPid() + ", " + per.getName() + ", "
80             + per.getAge() + ", " + per.getBirthday() + ", "
81             + per.getAddress());
82     }
83 }
84 public static void search() {
85     List<Person> all = null;
86     String kw = new InputData().getString("请输入检索的关键词：");
87     try {
```

```
88         all = DAOFactory.getIPersonDAOInstance().findAll(kw);
89     } catch (Exception e) {
90     }
91     Iterator<Person> iter = all.iterator();
92     while (iter.hasNext()) {
93         Person per = iter.next();
94         System.out.println(per.getPid() + ", " + per.getName() + ", "
95             + per.getAge() + ", " + per.getBirthDay() + ", "
96             + per.getAddress());
97     }
98 }
99 }
```

25.5.10 Test.java

写完这些代码之后，接下来可以进行测试，这时需要一个测试类。代码如下。

```
01 package org.lxh.info.test;
02 import org.lxh.info.menu.Menu;
03 public class Test {
04     public static void main(String[] args) throws Exception {
05         new Menu();
06     }
07 }
```



第 5 篇

王牌资源

实用、专业，这就是王牌。压箱底王牌倾情放送。

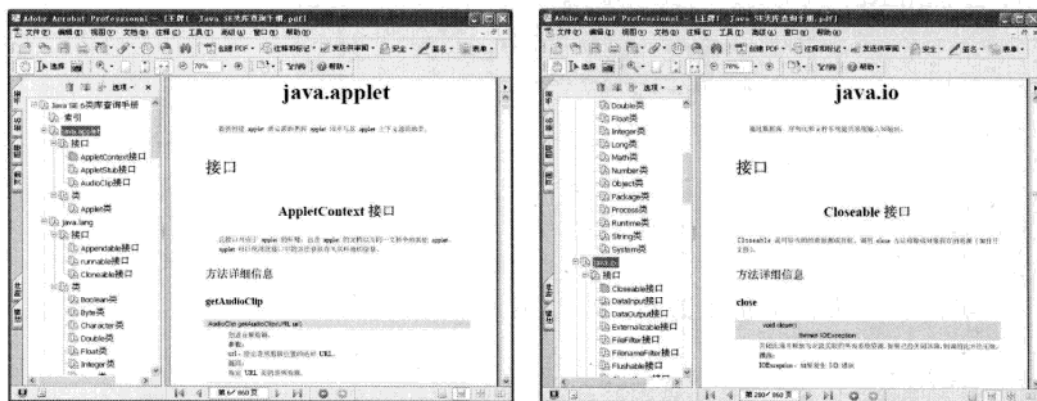
- ▶ 王牌 1 Java SE 类库查询手册（光盘中）
- ▶ 王牌 2 学习成果检测——本书【练一练】答案（光盘中）
- ▶ 王牌 3 Eclipse 常用快捷键（光盘中）
- ▶ 王牌 4 Eclipse 提示与技巧（光盘中）
- ▶ 王牌 5 Java 程序员职业规划（光盘中）
- ▶ 王牌 6 Java 程序员面试技巧（光盘中）
- ▶ 王牌 7 Java 常见面试题（光盘中）
- ▶ 王牌 8 扫雷英雄榜——Java 常见错误及解决方案（光盘中）
- ▶ 王牌 9 优秀程序员之路——Java 开发经验及技巧大汇总（光盘中）

王牌资源一览

王牌 1 Java SE 类库查询手册

位置：随书光盘\王牌资源\王牌 1 Java SE 类库查询手册.pdf

界面一览



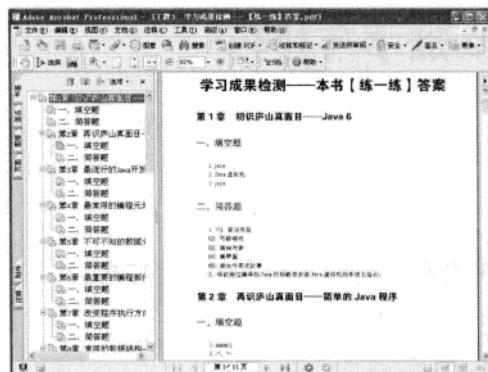
使用方法

- ① 在 PDF 阅读器中（如 Adobe Acrobat Professional）左侧的【书签】选项卡中显示了所有的类库列表。
- ② 在列表中单击需要查找的接口或类的名称，即可定位到此接口或类的详细说明处。

王牌 2 学习成果检测——本书【练一练】答案

位置：随书光盘\王牌资源\王牌 2 学习成果检测——本书【练一练】答案.pdf

界面一览



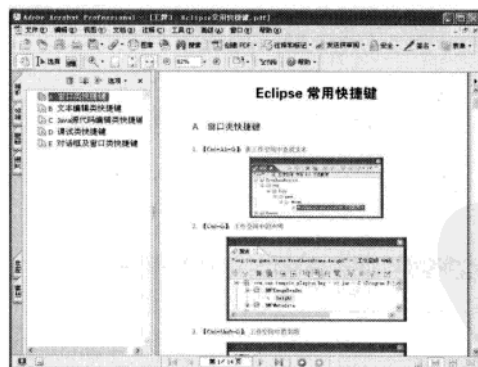
使用方法

- ① 在PDF阅读器（如Adobe Acrobat Professional）左侧的【书签】选项卡中显示了所有的章节答案的列表。
- ② 在列表中单击需要查找章节的名称，即可定位到此章答案的详细说明处。

王牌3 Eclipse 常用快捷键

位置：随书光盘\王牌资源\王牌3 Eclipse 常用快捷键.pdf

界面一览



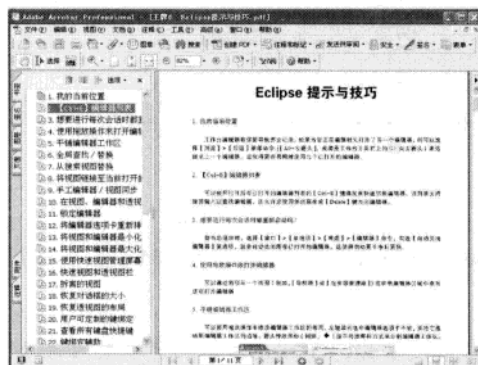
使用方法

- ① 在PDF阅读器（如Adobe Acrobat Professional）左侧的【书签】选项卡中显示了所有常用快捷键类的列表。
- ② 在列表中单击需要查找的快捷键类的名称，即可定位到此类快捷键的详细说明处。

王牌 4 Eclipse 提示与技巧

位置：随书光盘\王牌资源\王牌 4 Eclipse 提示与技巧.pdf

界面一览



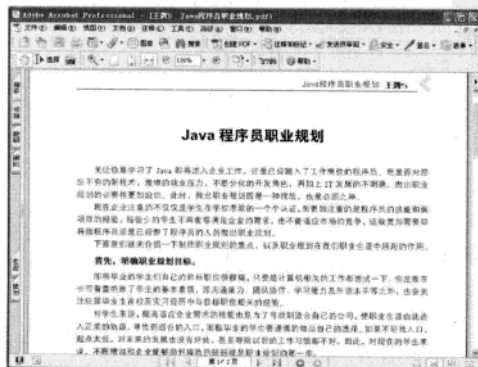
使用方法

- ① 在 PDF 阅读器（如 Adobe Acrobat Professional）左侧的【书签】选项卡中显示了提示与技巧的列表。
- ② 在列表中单击需要查找的提示或技巧的名称，即可定位到此提示或技巧的详细说明处。

王牌 5 Java 程序员职业规划

位置：随书光盘\王牌资源\王牌 5 Java 程序员职业规划.pdf

界面一览



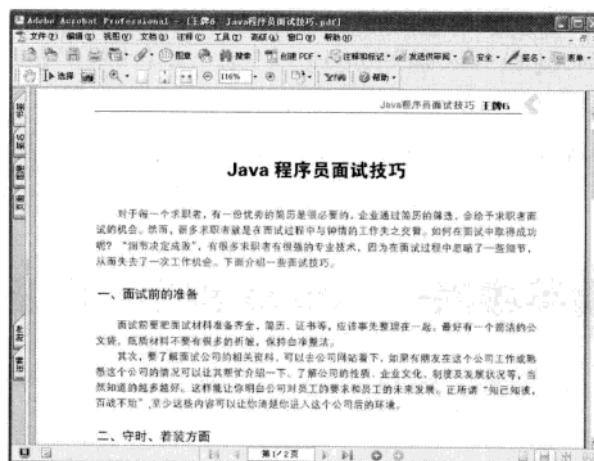
使用方法

可在 PDF 阅读器中（如 Adobe Acrobat Professional）阅读此内容。

王牌 6 Java 程序员面试技巧

位置：随书光盘\王牌资源\王牌 6 Java 程序员面试技巧.pdf

界面一览



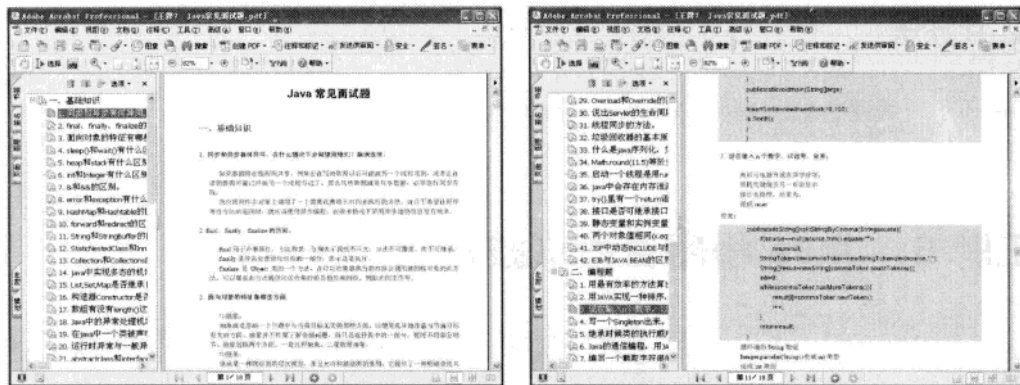
使用方法

可在 PDF 阅读器中（如 Adobe Acrobat Professional）阅读此内容。

王牌 7 Java 常见面试题

位置：随书光盘\王牌资源\王牌 7 Java 常见面试题.pdf

界面一览



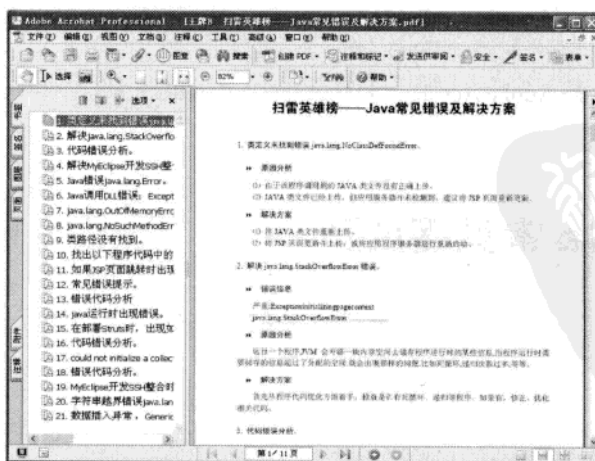
使用方法

- ① 在 PDF 阅读器（如 Adobe Acrobat Professional）左侧的【书签】选项卡中显示了所有的面试题的列表。
- ② 在列表中单击需要查找的面试题的名称，即可定位到此面试题的详细说明处。

王牌 8 扫雷英雄榜——Java 常见错误及解决方案

位置：随书光盘\王牌资源\王牌 8 扫雷英雄榜——Java 常见错误及解决方案.pdf

界面一览



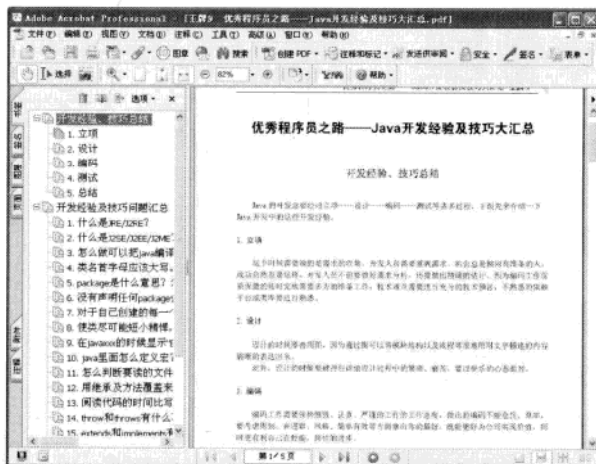
使用方法

- ① 在 PDF 阅读器（如 Adobe Acrobat Professional）左侧的【书签】选项卡中显示了所有的常见错误列表。
- ② 在列表中单击需要查找的错误名称，即可定位到此错误的详细说明处。

王牌 9 优秀程序员之路——Java 开发经验及技巧大汇总

位置：随书光盘\王牌资源\王牌 9 优秀程序员之路——Java 开发经验及技巧大汇总.pdf

界面一览



使用方法

- ① 在 PDF 阅读器（如 Adobe Acrobat Professional）左侧的【书签】选项卡中显示了所有的开发经验及技巧列表。
- ② 在列表中单击需要查找的开发经验或技巧的名称，即可定位到此内容的详细说明处。